August 2017

Efficient Implementation of Stochastic Inference on Heterogeneous Clusters and Spiking Neural Networks

Khadeer Ahmed
Syracuse University

ABSTRACT

Neuromorphic computing refers to brain inspired algorithms and architectures. This paradigm of computing can solve complex problems which were not possible with traditional computing methods. This is because such implementations learn to identify the required features and classify them based on its training, akin to how brains function. This task involves performing computation on large quantities of data. With this inspiration, a comprehensive multi-pronged approach is employed to study and efficiently implement neuromorphic inference model using heterogeneous clusters to address the problem using traditional Von Neumann architectures and by developing spiking neural networks (SNN) for native and ultra-low power implementation. In this regard, an extendable high-performance computing (HPC) framework and optimizations are proposed for heterogeneous clusters to modularize complex neuromorphic applications in a distributed manner. To achieve best possible throughput and load balancing for such modularized architectures a set of algorithms are proposed to suggest the optimal mapping of different modules as an asynchronous pipeline to the available cluster resources while considering the complex data dependencies between stages. On the other hand, SNNs are more biologically plausible and can achieve ultra-low power implementation due to its sparse spike based communication, which is possible with emerging non-Von Neumann computing platforms. As a significant progress in this direction, spiking neuron models capable of distributed online learning are proposed. A high performance SNN simulator (SpNSim) is developed for simulation of large scale mixed neuron model networks. An accompanying digital hardware neuron RTL is also proposed for efficient real time implementation of SNNs capable of online learning. Finally, a methodology for mapping probabilistic graphical model to off-the-shelf neurosynaptic processor (IBM TrueNorth) as a stochastic SNN is presented with ultra-low power consumption.

EFFICIENT IMPLEMENTATION OF STOCHASTIC INFERENCE ON HETEROGENEOUS CLUSTERS AND SPIKING NEURAL NETWORKS

by

Khadeer Ahmed

B.E., Visvesvaraya Technological University, 2006

M.S., Syracuse University, 2014

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical and Computer Engineering.

Syracuse University

August 2017

To my parents and my wife

ACKNOWLEDGEMENTS

I am grateful for the invaluable support from my advisor Dr. Qinru Qiu for guiding me throughout my research. She was always available to help whenever needed. Her inspiration to explore different avenues has helped me gain insights towards conducting research in a holistic manner. The sincere approach she takes towards understanding the problem has taught me to be meticulous and enabled me to make significant contributions. She has always motivated me to be a better researcher.

I would also like to thank various faculty who have helped me on numerous occasions especially Dr. Yanzhi Wang and Dr. Fawcett. My sincere gratitude to all my lab mates specifically Amar Shrestha, Syed Faisal and Zhe Li for their help and contributions in conducting my research. I am grateful to have wonderful friends who made my time at university pleasant and enriching.

Most importantly, I would like to thank my parents Fasiha Parveen and Mohammed Sharief for their hard work, encouragement and unwavering confidence in me. My wife Farah gave me the strength to bear the rigors of PhD and I appreciate her patience and moral support. It is your love and encouragement which made this possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

The brain has a very efficient and hierarchical architecture to process information [1]. It performs inference and decision-making tasks based on pattern matching and sensory association in the context of learned knowledge, which is the most important step towards cognition. With this motivation, the paradigm of brain inspired computing called neuromorphic computing has gained lot of attention recently. This emerging field of computing is offering a possible pathway for approaching the brain's computing performance and energy efficiency for cognitive applications such as pattern recognition, speech understanding, natural language processing etc. Currently more and more complex problems are being attacked using machine intelligence and deep learning concepts which fall under this paradigm. They are being adopted for Industrial applications and in research environment for solving problems which are very hard to articulate as it requires intuitive reasoning along with analytical abilities. Many strides in this field have been made from the inspiration of how the brain solves very complex problems using insights from well-established statistical analysis methodologies. These brain-inspired computing models have three main aspects to it; 1) the model itself which performs inference based predictions, 2) parameters used by the model to enable inference and decision making and 3) training the parameters for tuning the model to make better predictions. The ability of the model to learn is critical because of the vast parameter space one must explore, which is impossible to handle through the traditional programming approach. There are two main components which are the driving force behind these kinds of models. Firstly, large amounts of data and secondly, access to large amounts of compute resources to process it.

With the rapid development in high performance computing (HPC) technologies, the research

in machine intelligence has entered a new era. Meanwhile, modern computing systems are increasingly becoming more heterogeneous. This is due to a wide variety of computing architectures and accelerators such as multi-core CPU, GPU, FPGA, etc. being used. There are many questions which must be addressed for efficient utilization of these resources; how to harness the computing power and storage capacity of modern HPC clusters and convert it to useful computations that assist or even surpass the human cognition process? Will the performance of current neuromorphic computing models scale as the hardware resource increases? What is the bottleneck of current HPC architectures when applied to cognitive computing and how can this be addressed by future computing tools? This work makes a preliminary effort in answering these questions. We propose a framework to implement complex applications as pipelined distributed applications which are capable of seamless scaling over heterogeneous cluster resources. It is also capable of distributed flow control and dynamic task dependency aware scheduling. Next, we address the problem of efficient resource allocation for such complex systems. We use a complex neuromorphic application, Intelligent Text Recognition System (ITRS) [2], as a case study to validate the framework and discuss the major advances along different modalities. The background of such a system along with key algorithms will be discussed. Insights into designing modular pipeline stages for complex applications which scale with the available compute resources is provided. We do a comparative analysis of our framework with existing solution for demonstrating the effectiveness of the proposed approach.

On the flip side, the exponential growth of data over the past decade has generated a need for higher processing capability with low energy consumption and ease of scalability. Limitation of the Von Neumann architecture and barriers such as memory capacity, power density etc. in the CMOS technology are being highly tested to meet today's requirements and also to fulfill

Moore's predictions. These limitations have motivated novel research efforts in bio-inspired computing, which imitates the structure and function of the brain, the computing engine that is able to process massive amounts of real-time information with less than 20 Watts of power consumption [3]. The processing capability of brain comes from the collective processing abilities of simple processing components i.e., neurons. Interconnected neurons form the basis of a neural network. The ability of neural networks to perform pattern recognition, classification and associative memory, is essential to applications such as character recognition, speech recognition, sensor networks, decision making etc. [4] [5] [6] [7] [8]. SNNs, which use spikes as the basis for communication, are the third generation of neural networks inspired by the biological neuron models [9].

The SNN has the potential to reach very low energy dissipation since each neuron works asynchronously in an event-driven manner. Moreover, fully distributed Spike Timing Dependent Plasticity (STDP) learning [10] can be achieved on SNNs, which updates synaptic weight based only on local information of individual neuron. The emerging field of stochastic SNN that generates spikes as a stochastic process is not only more biologically plausible [11] but also enhances unsupervised learning and decision making [12] [13]. It further increases the fault tolerance and noise (delay) resilience of the SNN system since the results no longer depend on the information carried by individual spikes but the statistics of a group of spikes. With this inspiration a focused effort is made to present a high-performance SNN simulation framework for developing spiking neuron models and simulation of large-scale SNNs. For efficient and native implementation, digital hardware is proposed for SNN implementation. Finally, a methodology is presented for modelling probabilistic graphical model used in ITRS as SNN is presented, which is realized on off-the-shelf neurosynaptic processor for ultra-low power and

real-time evaluation of the neural network.

1.1 CONTRIBUTIONS

To address the challenges of implementing efficient brain inspired systems different approaches have been adopted spanning system level design decisions to low level optimizations. The primary contributions of this work are listed below

1. A Scalable Linear Pipeline (SLP) framework is proposed which integrates the optimization techniques for node level system design and cluster level distributed system design for a holistic approach using existing computing technologies.
2. The concept of scalability of a pipeline is introduced. Each stage is made modular with uniform communication architecture for flexibility of mixed module designs which allow for maximum available resource utilization without the need for expensive application redesign for heterogeneous clusters.
3. Asynchronous pipeline concepts are introduced for such a scalable architecture which enables out-of-order computation, hence minimizing idle time i.e. increased throughput.
4. Novel structure based runtime scheduling is introduced for achieving maximum performance for asynchronous workload processing with varying module latencies while respecting the data dependencies.
5. For achieving best possible throughput, a set of algorithms are proposed to suggest the mapping of software modules to various hardware resources available on a heterogeneous cluster.
6. For a more efficient and biologically plausible brain inspired implementation, several spiking neuron models are proposed which are capable of distributed on-line learning.

7. An efficient and high-performance spiking neural network simulator architecture is put forward for large-scale SNN simulation involving mixed neuron models and different learning rules.
8. A digital spiking neuron hardware design is proposed which is capable of online learning for a comprehensive take on SNN implementations. The architecture is pipelined to compute inference and learning task with approximately same throughput compared to existing digital spiking neuron model implementations including those which don't implement in hardware learning.
9. Finally a streamlined approach is presented to map a probabilistic inference model as a spiking neural network on existing off-the-shelf neurosynaptic processor

2 SCALABLE LINEAR PIPELINE FRAMEWORK

Today increasingly complex applications are being moved from the end user to the cloud infrastructure [14] [15], due to the cost advantage and ease of access to large amounts of computing resources. This shift has positively impacted the field of research, scientific computing, big data, large scale consumer applications, complex system simulations, neuromorphic computing, financial modeling etc. The key enablers for this shift are the reducing cost of high performance computing (HPC) resources and the ability to handle large amounts of data [16]. Distributed data storage and management techniques have become very popular to sieve through large amounts of data efficiently [17] [18]. Tremendous technological advancements are being made in terms of computing accelerators, resulting in the rapidly increasing popularity of heterogeneous clusters. Since these clusters include processors with different basic architectures, they provide unique performance and cost tradeoffs for different types of workloads. To achieve peak performance, software running on heterogeneous cluster needs to be designed carefully to provide enough flexibility to explore its diversity. With these developments in HPC technologies, the design and development of high impact, complex applications, especially in the field of machine intelligence have entered a new era. Since these clusters include processors with different basic architectures, they provide unique performance and cost tradeoffs for different types of workloads. To achieve peak performance, software running on heterogeneous cluster needs to be designed carefully to provide enough flexibility to explore its diversity.

These applications are designed as linear pipelines to maximize their throughput as they process huge amounts of requests in a streaming manner while requiring access to large amounts

of data. They have the ability to hide the overhead of managing communication, processing and synchronization which are very beneficial for HPC paradigm [19]. Significant research has been made in modeling such applications especially in the context of large-scale platforms. There are many challenges including the design of applications, identifying different stages of the pipeline, identifying a suitable pipelining model, data partitioning, parallelizing, mapping of pipeline stages to different resources etc. Traditionally Linear pipeline models are preferred due to their simplicity in design and implementation. For streaming applications, all stages of the pipeline must be active and processing requests hence requiring more resources compared to non-streaming applications where interval based resource allocation is a standard. These linear pipeline models are limited in terms of their scalability. To scale these models, it requires a fresh look at how the stages can be re-decomposed to improve the performance. In this work, we propose a *Scalable Linear Pipeline* (SLP) framework which overcomes these limitations and affords seamless scalability over a heterogeneous cluster while performing dynamic distributed load balancing, distributed flow control along with data dependency aware scheduling. In a heterogeneous cluster a stage can be mapped to run on only a limited number of nodes, making the problem of mapping the pipeline harder. It is a non-trivial task to determine a mapping for such a highly-constrained model as in this model we allow simultaneous compute resource sharing for stream processing which is more desirable in the real-world applications.

We build the SLP framework using principles of traditional lineal pipeline model. The task dependencies spanning across different stages is modelled as a dependency graph and how their behavior scales with the increasing number of resources is outlined. The SLP allows automatic load balancing and self-scheduling. These capabilities are explained using the dependency graph and a simple analytical performance estimation model is presented. The SLP also allows flexible

resource utilization. The algorithm for mapping the SLP to available resources in a heterogeneous cluster is discussed.

Distributed systems are widely used and have been extensively studied. Different kinds of distributed pipeline based architectures are proposed. A state based distributed pipeline framework is presented in [20]. Here the compute nodes are separated from the pipeline control. Instead of message passing the state objects are passed which encapsulate the data. The load balancing is achieved through producer/consumer relationship i.e. processing happens asynchronously. However, there is an extra overhead in creating and decoding state objects at every stage apart from data processing. A distributed pipeline processing architecture composed of flow-models, called meta-pipeline is proposed for general-purpose computation [21]. The architecture is suitable for stream based processing. This requires input and output streams along with the parameters for every flow-model. These details and other properties are encapsulated in XML. This kind of modularization enables distributed task based execution. Though this system is distributed it requires centralized management to assign and load flow-models. Fully utilizing the performance of heterogeneous resources is a challenging task. Design methodology for executing applications on heterogeneous platforms, which are specified as synchronous dataflow (SDF) graphs is proposed in [22]. The authors try to maximize the end-to-end throughput of an application developed in OpenCL by modeling it using SDF graph. Data-intensive workflow optimization is presented in [23] which uses task graph partitioning to improve the performance of streaming applications on heterogeneous systems. By minimizing the data movement between partitions, they reduce the latency and increase the overall throughput, however they allow task duplication across partitions. This method is not suitable for applications which have dynamic task dependencies.

Mapping such pipelined applications to compute resources is a non-trivial task. The work presented in [24] discuss the theoretical aspects of a linear pipeline with computation and communication overlap. They present models for interval based resource mapping for homogeneous and heterogeneous platforms. The ELPC model presented in [25] discusses mapping of linear pipeline models over a wide area network. They present a dynamic programming approach to solve the mapping problem. A similar approach is demonstrated in [26] with a focus on visualization pipeline. In the above works the mapping is done with interval based resource sharing. This introduces additional complexities on optimizing buffer sizes which is studied in the work presented in [27]. These approaches are not very scalable as the pipeline design has to re-worked to improve the performance. These pipeline models are developed for distributed applications. They also don't address the problem of task scheduling for complex workloads which is partly because their models don't scale easily hence having very simple first come first serve scheduling. A cuckoo search based scheduler for mapping of workflow tasks on heterogeneous cloud resources is presented in [28].

Clustering Method based Task Dependency resolution scheme is introduced in [29] to handle today's complex data dependent task scheduling in distributed application environments. In this task clustering methodology, they merge fined-grained tasks into coarse-grained jobs. Hence, with clustering they try to reduce execution overhead and to improve the computational granularity on distributed resources. This approach, requires the detailed knowledge of tasks which differ significantly for different applications and also needs a centralized analysis to determine the scheduling behavior. Ke Wang et al. [30] present a locality aware load balancing scheme for data-intensive workloads many-task computing models. This is a fully distributed task scheduling architecture however; this approach requires full connection among compute nodes

which is not always ideal. A k-means algorithm based initial data placement strategy [31] is introduced to have optimized task scheduling for data intensive workloads for distributed applications.

In contrast to the aforementioned related works, the goal of our work is to show how complex application with various processing requirements can be converted to scalable, distributed applications even for data intensive workloads. The research community has been focusing on different aspects of pipeline processing and strategies for scaling applications in distributed environments as different problems. This is the first work to our knowledge where we introduce the concept of scaling of pipelined application models over distributed resources. We also introduce a distributed scheduling strategy for such implementations which enables dynamic load balancing and out-of-order execution for higher hardware resource utilization. Compared to a traditional linear pipeline optimized using the Efficient Linear Pipeline Configuration (ELPC) [25], the SLP and our proposed resource mapping algorithm achieve the resource utilization efficiency, which was not possible before. To validate our performance estimation model and to demonstrate the potential of the SLP, we implement a neuromorphic application called Intelligent Text Recognition System, as a case study. Details of pipeline construction, management, scheduling and inter-stage communication are discussed. The experimental results show that our performance estimation model achieves 96% accuracy compared to the measured results, and our resource mapping algorithm improves resource utilization and is capable of providing linear scaling where ELPC failed to address these concepts.

2.1 PIPELINE MODEL

For any streaming application, the end-to-end throughput is among the essential factors

considered during the design phase. Latency of such applications play an important role however, it is not critical while determining the sustained throughput. Large applications, requiring heavy computation while needing access to large databases are typically implemented as deep pipelines. Pipelines are efficient for streaming applications as they have the ability to hide latencies across various stages, while trying to optimize for maximum resource utilization. In general, complex applications are looked as a linear sequence of distinct stages. Each stage in a linear pipeline performs a task in-order, hence they are referred to as first-in-first-out systems.

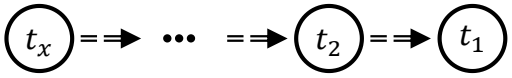


Fig. 1. Typical task dependencies of a linear pipeline

Various graph based models have been used to capture task dependicies. A workflow graph representing linear task dependencies which can be resolved in-order fashion is shown in Fig. 1. Each node in this graph represents a computation task and the directed edge represents the data dependency. For example, t_2 depends on t_1 and so on until the final task t_x which depends on its previous task t_{x-1}. It is important to note that the task dependency is known at design time and it specifies the task execution order. We assume that ASAP scheduling is adopted, which will start the execution of a computing task as soon as all of its inputs are ready and the computing resource is available. If multiple tasks are ready for execution, then the earliest one will be picked.Fig. 1. Each node in this graph represents a computation task and the directed edge represents the data dependency. For example, t_2 depends on t_1 and so on until the final task t_x which depends on its previous task t_{x-1}. It is important to note that the task dependency is known at design time and it specifies the task execution order. We assume that ASAP scheduling is adopted, which will start the execution of a computing task as soon as all of its inputs are ready and the computing resource is available. If multiple tasks are ready for execution, then the earliest

one will be picked.

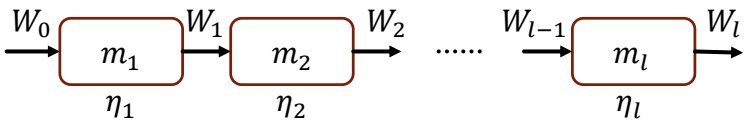


Fig. 2. Linear pipeline model

The above linear dependency directly corresponds to a *Linear Pipeline* (LP) model. Fig. 2 shows a generic block diagram of such a linear pipeline. It has l stages. The output W_k of any stage m_k where $1 \leq k \leq l$ is considered as the workload for subsequent stage m_{k+1}, here W_0 is the input to the pipeline and W_l is the final output of the pipeline. Each workload represents a single task for the next stage. Each stage of the pipeline can resolve one task or a sequence of consecutive tasks in the workflow graph. After modeling the pipeline, we map each stage to a computing resource. To achieve the highest parallelism and maximum resource utilization, it is desirable to have similar latency in each stage hence no computing resource is idle during the processing.

We focus our attention towards streaming pipelines as they are more practical in processing large datasets by taking advantage of concurrent stages. Especially when it is required to look up large distributed databases which is common in today's applications such as; neuromorphic applications, business intelligence, big data, machine learning, deep learning etc. The only way to scale such pipelines is to break the tasks at a finer grain and make the pipeline deeper, i.e. add more stages to increase the effective parallelism. However, this is limited by the granularity of hardware computing resources and the parallelism within a computing task.

A traditional linear pipeline as shown in Fig. 2 is primarily a first-in-first out system which doesn't scale efficiently. Making the pipeline deep is the only way of scaling which has a significant overhead in terms of remodeling the tasks and redesigning the computing stages to

scale such pipelines. To overcome the pipeline scaling issue, we propose a *Scalable Linear Pipeline* (SLP) framework. The SLP looks at the application at the cluster level and treats it as a macro pipeline. To achieve best performance every module must be efficient enough to keep this macro pipeline busy as much as possible. To achieve this goal, we treat each module as a micro pipeline. Each stage is treated as a pipeline for achieving higher throughput but this is not a binding requirement by the framework. We employ a modular approach to enable scaling not only along the number of stages but we scale at every stage, hence resulting in wider pipeline. A typical SLP is shown in Fig. 3.

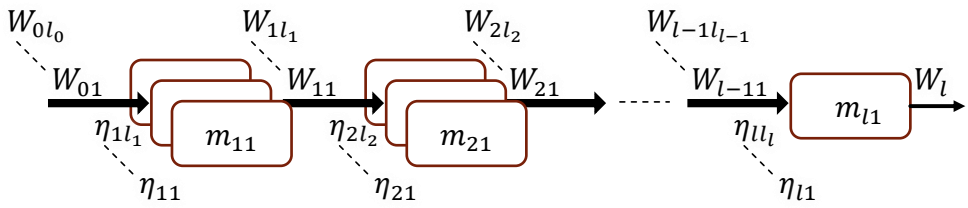


Fig. 3. Scalable linear pipeline model

In contrast with a LP, SLP consists of consecutive layers composing a linear pipeline as each layer consists of multiple parallel instances of a stage. Therefore, SLP consists of l layers. Each layer $k, 1 \leq k \leq l$ is a set of parallel stage instances m_{ki} called as modules, where $i \geq 1$ is an arbitrary number based on the scaling of the k^{th} layer. Multiple independent tasks are grouped as workload to minimize traffic between modules and reduce the compute resource idle time at the destination module. The result W_{ki} of any module of a layer l is considered as the workload for subsequent layer $l + 1$, here W_{0i} are the input workloads to the pipeline and W_l is the final output of the pipeline.

In SLP, tasks in the workflow graph are partitioned such that there are no data dependencies across workloads of the same layer. The modules in the same layer operates asynchronously.

There is no guarantee of the order of the task completions. This will not cause problem because the task precedence constraints are ensured by the structure of the pipeline and the ASAP scheduling of resolved tasks. Due to scaling of a layer a task can have dependencies from different modules of previous layer. Any unresolved task is buffered to enable computation of resolved tasks in out-of-order manner. Even with such task dependency, this model is a linear pipeline as the dependencies are between consecutive layers. The last layer of SLP has only one instance, which collects and reorder the out-of-order completed tasks.

Each module can have a variety of design requirements based on the application hence, it is not practical to build a model which suits all the design decisions of distributed applications. Instead we outline few strategies which are ideal for SLP based implementation. An input task can be further parallelized to sub tasks and scheduled to thread pools or multiple whole tasks can be scheduled to thread pools for computation on a CPU based architecture. The computation can be vectorized or can be accelerated and optimized based on the hardware platform requirements. The key idea behind these suggestions is to keep the hardware resources busy as long as possible while maintaining out-of-order computation which perform task resolution based scheduling. For these micro pipelines to work together a common communication interface which runs independently and in parallel to computation is needed to make the system design modular so as to scale seamlessly. Therefore, SLP is a pipeline of pipelines working asynchronously.

The proposed SLP is flexible as it is agnostic to the nature of computation that happens within an instance of a stage as long as the input and output workloads are of same type as in that layer. Therefore, the model supports modular design for implementing the pipeline as modules in a layer can be implement on different hardware platforms and technologies with different latencies. A typical task dependency graph for SLP is shown in Fig. 4 for three consecutive layers. Here

task t_{1k} of layer k has dependence on a set of tasks t_{1k-1} up to t_{xk-1} of previous layer $k-1$. From the figure, hierarchical task dependency is evident.

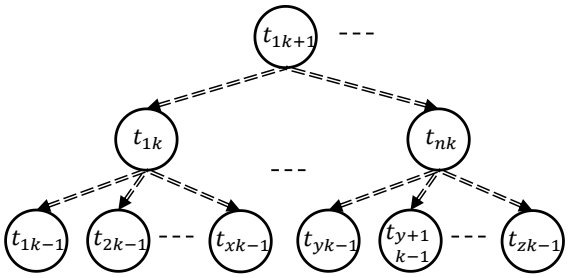


Fig. 4. Typical task dependencies of a scalable linear pipeline

Task data dependencies are application and data specific. In some applications where bottom up processing is used, the output of multiple computing tasks at fine granularity will be assembled as one workload to trigger higher level processing. In other applications where top down processing flow is used, the output of one upper level computing task will trigger multiple lower level computing tasks. Although both are interesting scenarios in the SLP optimization, the latter is more challenging in the SLP, which performs out-of-order-execution, because it consists of tasks with multiple dependencies. The number of upper or lower level tasks and their relations are often not fixed, but rather data driven. To support out-of-order computation we allow breaking linear tasks to fine grained sub-tasks when compared to LP model. This procedure creates dependency as the actual task is resolved only when the sub-tasks are resolved. The number of sub-tasks is dependent on the size of the actual task which is dynamic. These sub-tasks are treated as independent tasks in SLP, which are processed by employing fork and converge strategy. As the tasks across the layers converge downstream their dependencies get resolved and the processing gets done hierarchically. Therefore, actual task dependence graph cannot be known at design time. The first layer forks the tasks, the middle layers converge hierarchically to resolve the dependencies and the last layer is to re-order the out-of-order resolved tasks.

Therefore, SLP requires a minimum of 3 layers to support the fork and converge design methodology. The number of upstream layer tasks required to resolve one task in the current layer cannot be known at design time. Such information is only available during runtime. However, the general fork and converge structure is known at design time. Using this knowledge structure based runtime scheduler is proposed to resolve these dynamic task dependencies. We discuss this in section 2.5.

In the proposed design methodology, the communication happens asynchronously and in parallel to computation. Therefore, the communication latency is hidden. Let L_{sc} represent the link delay between a given pair of adjacent layer modules (m_s, m_c) where subscript s stands for source and c stands for consumer. Since the latency is hidden L_{sc} is not a critical parameter in our model. For SLP to guarantee deadlock free operation the input of every module must have input buffers large enough to accommodate partial results from previous layers till the task of that layer is resolved. It must have output buffers as well to temporarily store the results from the micro pipeline till the results can be forwarded to the next downstream module. If the output buffer is full then it stalls the micro pipeline, till there is room to store new results. However, it is straight forward to compute the minimum size of input and output buffers of a module by accounting for the variance in the rate of messages received and the rate of messages processed. In this work, we target a computing cluster, more specifically a heterogeneous cluster instead of a wide area network of computing resources. Therefore, the latencies L_{sc} are small. Our model can be easily extended to a wide area network scenario with larger input and output buffers to account for the variability of message arrival rates and link latencies.

2.2 DYNAMIC DATA DEPENDENCY

To accommodate scaling at layer level, the modularization strategy was based on breaking complex tasks into multilevel data dependencies and creating large number of small independent workloads which can be processed parallelly by different modules in a layer. This results in dynamic data dependencies which need to be resolved in real time. The first layer creates small independent tasks which are forked to the next layer hence, there is one-to-one task dependency with the second layer as shown in Fig. 5(a). The dependency is one-to-one because bottom-up approach is used and the first layer created the leaf tasks which must be now processed in the next layer.

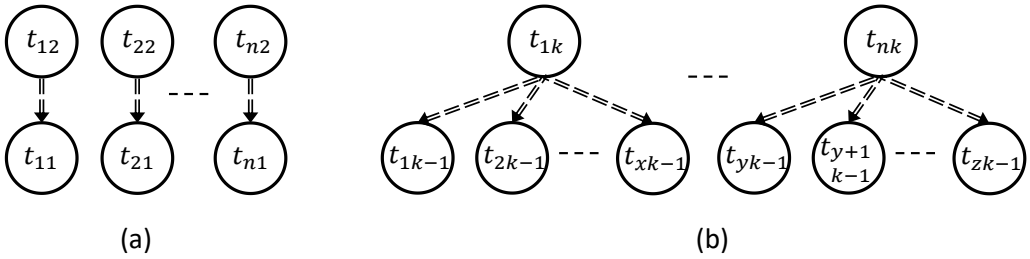


Fig. 5. (a) Task dependency between 1st and 2nd layer (b) Task dependence between consecutive layers except 1st and 2nd layer

All other consecutive layers have convergent task dependence as shown in Fig. 5(b). A workload is a group of tasks which is transmitted as messages from one module to other across consecutive layers over point to point links. Since all tasks in a layer are independent, all tasks with in any given workload are mutually exclusive. However, those tasks have dependencies across layers. Depending on the resources available and the input these tasks complete asynchronously. Task t_x at the k^{th} layer is resolved if the set of dependent tasks Γ_{xk-1} from the previous layer in the graph are computed which is denoted as

$$X(t_{xk}) = \wedge_{\forall t_i \in \Gamma_{xk-1}} (X(t_i))$$

The results of the subset of tasks belonging to Γ_{xk-1} which are not yet resolved must be buffered at the input of every module until $X(t_{xk})$ is resolved. Once $X(t_{xk})$ is resolved, task t_{xk} is scheduled for computation. The rate at which $X(t_{xk})$ is resolved depends on the input and the compute resources available, therefore it is critical to have multiple such tasks queued up to increase resource utilization. This condition can be met by having large number of tasks in the workloads and having multiple redundant paths of execution in the scaled pipeline which is determined by the width of the pipeline at that layer.

To support such task resolution in a de-centralized manner we apply constraints on the connectivity pattern among modules between every consecutive layer. We use point-to-point connectivity between modules of consecutive layers to keep the data flow de-centralized. Two types of connectivity patterns are used to support the fork and converge model described earlier as shown in Fig. 6.

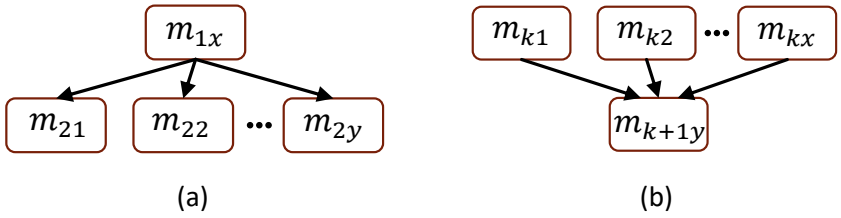


Fig. 6. (a) Single fan-in, multi fan-out connectivity (b) Multi fan-in, single fan-out connectivity

Fig. 6(a) shows the connectivity constraints between the first two layers of the pipeline, it has one-to-many connectivity (C1M) pattern. Fig. 6(b) shows the connectivity constraints between all consecutive layers except 1st and 2nd layer. This pattern has many-to-one connectivity (CM1). For the case of C1M the module m_{1x} breaks up its workload into small tasks and schedules it to one of its out-going paths thereby performing a fork operation. CM1 on the other hand performs the converge operation by reducing the results from several upstream tasks. Since CM1 is present

between many layers, the reduction happens hierarchically. Therefore, the connectivity constraints which help in determining the number of source and consumer modules is expressed as

$$\text{C1M constraint: } |m_s| \leq |m_c|$$

$$\text{CM1 constraint: } |m_s| \geq |m_c|$$

We try to match the performance of each layer by managing the level of parallelism in each layer. The the number of modules required in each layer is determined to achieve the required performance, we refer to this as scaling. After determining the scaling of each layer, these modules are interconnected based on the connectivity constraints. The resulting pipeline graph is called as *System Topology Graph* (STG). Fig. 7 shows an example of a typical STG for the SLP. This example has 5 layers with 2,5,3,2,1 scaling in layers 1 through 5 respectively. It is interesting to note that SLP is a super-model of LP model. If we restrict one module per layer and restrict that each workload is one task then SLP reduces to a LP model.

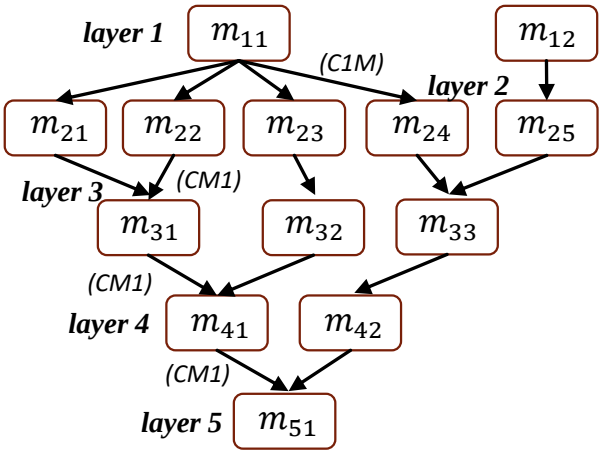


Fig. 7. A typical system topology graph

2.3 PERFORMANCE MODEL OF SLP

Every module of the k^{th} layer, $m \in m_{k1}, m_{k2}, m_{k3}, \dots, m_{ki}$ can run with different configurations resulting in different throughput $\eta \in \eta_{k1}, \eta_{k2}, \eta_{k3}, \dots, \eta_{ki}$. Those configurations include the number of threads in the software implementation, the assignment of hardware platform, or other algorithm based settings. Different binaries can also be developed to run the same module on different hardware architecture. These binaries are individual processes represented as $p \in p_1, p_2, p_3, \dots, p_u$ where u is the maximum number of processes for any given module m. Therefore, $c \in c_1, c_2, c_3, \dots, c_n$ represent configurations where, n is the maximum number of unique set of process-parameters associated with each given process p of a module m. In other words, a module m can be realized by any of c_1 to c_n configurations. Where each configuration represents a process binary for a given hardware architecture along with the associated parameters. Each layer can run a mixture of these configurations hence SLP has a large design space.

It is a common practice to model the processing performance of different hardware nodes used in the system to be normalized across performance of different modules. This abstraction encapsulates the processing speed, memory, bus speed etc. and makes mapping algorithms simpler however, this adds an approximation to the model. Instead of modeling the system using normalized processing requirements for modules and the available hardware resources, we perform dry runs to collect empirical data for reliable performance modeling. However, this is not a limiting factor, as the same model can be used with normalized representation of performance requirements. For every server $s \in s_1, s_2, s_3, \dots, s_r$ with r nodes in the heterogeneous cluster, let $T_{max,s}$ denote the number of logical cores in s, and $M_{max,s}$ its peak memory bandwidth. This gives us the upper limit of the supported compute and memory bandwidth capacity for every

server s.

Next, we determine the CPU level thread concurrency T and the memory bandwidth M needed to run every configuration c of each module along with its run time τ. It is necessary to measure the CPU level concurrency as each module configuration consists of threads performing different tasks which are not always concurrent, this way we obtain the actual impact of the configuration on specific compute resource. To enable this data collection, each process along with its set of parameters must be analyzed independently i.e. $\forall c \in m$. The value of τ recorded is the run time achievable for the given configuration c. The value of τ is normalized to the unit task the end-to-end pipeline is processing to get per unit work of runtime. T and M are not normalized as they represent the steady state requirements to run a configuration. Rate is computed as $\eta = 1/\tau$ which represents the number of unit work processed per second for every $c \in m$.

For SLP to run at maximum performance all modules must be processing at maximum capacity. This is possible when every module is receiving tasks at maximum input rate. Therefore, to determine the performance of such a pipeline we measure the performance of individual modules in a standalone manner for maximum input rate. Using the standalone performance as building blocks, we determine the SLP performance. In practice, it is not practical to assume that all the modules have same performance. The performance of SLP is determined by the layer with least throughput, i.e. max runtime. Therefore, we model the throughput η_P of SLP as

$$\eta_P = min_{k=1\ to\ l}(\sum_{i=|m_k|} \eta_{ki})$$

Mapping hardware resources to run different modules and to determine the STG for such mapping while optimizing the end-to-end throughput is a non-trivial task. We provide the solution for this challenge in section 2.4 based on the performance model discussed here.

2.4 RESOURCE MAPPING FOR MAXIMUM THROUGHPUT

To achieve high performance, all pipeline stages should have the same throughput. However, the workloads of different layers differ significantly. A layer with heavy load should be able to grab more computing resources and scale accordingly. Each software module which runs on a hardware node can employ multi-threading or any hardware platform specific acceleration and optimization to achieve maximum efficiency possible. The performance of a module (configurations c) and the number of modules in a layer are parameters that are determined to keep a balanced pipeline. To allocate more resources to a particular layer, we simply need to instantiate more modules or use different configuration of a module of that layer. In a heterogeneous system, their selection not only depend on the layer a module belongs to but also the hardware that the modules and its configurations that can run on it.

The goal of resource mapping is to find the best SLP structure and a mapping between SLP modules and hardware computing resources to achieve optimum throughput. During this procedure, we add or remove SLP modules to balance the throughput among layers, therefore the structure of SLP and the mapping scheme evolve simultaneously. Please note that in a heterogeneous system, maximum resource utilization does not necessary mean maximum throughput.

Resource mapping for maximum performance is a hard combinatorial problem. Our heuristic algorithm consists of two major steps. First, we find a *minimum feasible solution* (MFS) such that one module from every layer is assigned a compute resource. Then we improve the MFS by allocating additional modules to available compute resources to eliminate bottlenecks and achieve a desired throughput. The throughput of every module and end-to-end throughput of the pipeline is measured in terms of number of unit-work processed per second.

Many constrained resource matching problems are solved using dynamic programming, which has pseudo-polynomial time complexity. To apply dynamic programming, we must be able to construct the optimal solution of the problem based on the optimal solution of its sub-problems. This requires the solution space to be discretized resulting in re-use of sub-problem solutions. The work presented in [25] solves a resource mapping problem for a linear pipeline using dynamic programming techniques. They propose an algorithm called Efficient Linear Pipeline Configuration (ELPC) with a constraint that a hardware resource is not concurrently running multiple modules while optimizing for end-to-end throughput. ELPC however allows interval based resource sharing with different modules which is again non-concurrent sharing. In the proposed pipeline model each module has multiple configurations who are candidates for resource mapping to hardware resource which is already mapped with a configuration which has partially utilized that resource. This kind of mapping improves resource utilization efficiency. Therefore, the proposed model is more efficient than ELPC as it tries to utilize the hardware resources to the maximum extent possible. Since SLP allows simultaneous resource sharing, the sub-problems of partial resource allocation can't be guaranteed to have optimal solution due to fractional allocation of resources. The sub-problem solution can't guarantee optimal sharing of a resource till all the configurations of not yet visited sub-problems are analyzed. Therefore, we propose a solution based on backtracking methodology.

For solving the mapping and throughput optimization problem we make use of a resource allocation graph with some enhancements to keep track of resource sharing, we call this *Simultaneous Resource Allocation Graph* (SRAG) as shown in Fig. 8. Every edge represents a configuration c of a module m. A request edge represents a resource allocation request from a module to a hardware resource. The assignment edge represents a mapping between module and

hardware. We introduce another type of edge called invalid edge which is represents a configuration that was deemed infeasible for mapping based on the available resources. Therefore, to run a module there are a set of associated configurations which can be allocated to hardware resources based on their availability in the cluster. Each configuration has an associated cost in terms of required concurrency, memory bandwidth, number and type of accelerator cards etc.

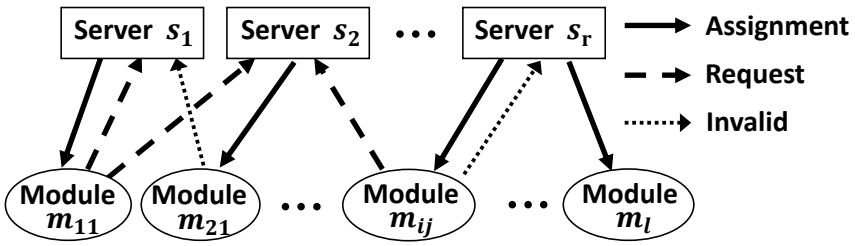


Fig. 8. A typical simultaneous resource allocation graph

We now introduce the properties and methods of SRAG. The SRAG is used to keep track of module assignments. It evolves iteratively till a final solution is obtained. Each iteration consists of updating an edge state of SRAG which involve setting an edge type as assignment, request or invalid. A request edge of SRAG is transformed to an assignment edge if the hardware has enough resources available as required by the cost of the edge, if not than this edge is transformed to an invalid edge. Whenever an edge is transformed from request or invalid type to assignment type then the edge-cost amount is deducted from the available resources for that server, indicating the amount of hardware resource used up for this assignment. Inversely, when an edge is transformed from assignment type to request or invalid type then the edge-cost amount is added back to the available resources of that server, indicating freeing up of hardware resources.

An evolution of SRAG is defined as a set of iterations of SRAG which result in assigning minimum number of modules which results in an increase of overall throughput. After every

evolution of SRAG the corresponding STG is computed. It is important to note that a module can have only one assignment edge associated with it as it can run on only one server at a time. In contrast, STG consists of connections between the modules which represents the actual system topology of the modularized distributed application which runs on the heterogeneous cluster.

2.4.1 MINIMUM FEASIBLE SOLUTION

Initial SRAG is created with one modules each for a layer. It also has as many resource vertices as the number of available servers. The SRAG at this point has only request edges, which represent all possible configurations c to work with. The MFS algorithm is a recursive function based on backtracking principles. Backtracking performs exhaustive recursion which can be potentially very expensive. For every recursive call, we make a decision-point for getting a feasible assignment and continue further to explore next feasible assignment. If further such assignment is not possible then we backtrack to the decision-point and try other alternatives. In this way, we backtrack only as far as needed. We apply a heuristic by pre-processing the input to the algorithm to reduce the recursion depth for average case.

A list of request edges is made by selecting one edge per module from the request edges belonging to each module. The selected edge has minimum run time among all the request edges of that module. While comparing a tie on run time is broken with the edge having minimum memory bandwidth and a tie on this is further broken with the layer priority of the module. Modules in layer 1 have highest priority and modules of layer l have the least priority. This list represents the best possible assignment each module can potentially get. It is logical to map upstream modules before the downstream ones so that potential bottleneck may appear in lower layers hence reducing future optimization effort. For this reason, layer priority is used as a tie breaker. These selected edges are now sorted with the same comparison policy but in descending

order. Therefore, the edge with highest runtime is on top in the sorted edge list. We now make a two-dimensional jagged array (modEdgeLists mel) with each row containing all the request edges of a module. The order of rows of this array is same as the module order associated with the sorted edge list. The row order represents the possible bottleneck layer hence this module will be mapped first. Each column of a row in mel represents the possible configurations the module represented by that row can have. The elements of every row are sorted in ascending order with the comparison policy mentioned above. From this ordering of mel we can say that; potential bottleneck layer is assigned its best configuration first. Ordering of mel which is the adjacency list of SRAG constitutes the pre-processing of initial conditions to the algorithm.

Algorithm 1. Minimum Feasible Solution

```
Function MFS
Input: modEdgeLists mel, module index mIdx
Output: feasible edge assigned
for each edge e ∈ mel[mIdx] do
    if e=type Request then
        if assignment of e is possible then
            set e type ← Assign
            edgAssigned ← true
            if mIdx ≠ l then
                mIdx ← mIdx + 1
                if MFS(mel, mIdx)=false then
                    set e type ← Invalid
                    edgAssigned ← false
                    mIdx ← mIdx − 1
            If edgAssigned = true then
                break
        else
            set e type ← Invalid
if edgAssigned = false then
    for each edge e ∈ mel[mIdx] do
        set e type ← Request
    return false
return true
```

2.4.2 SLACK BASED TOPOLOGY CREATION

Each module vertex has 3 parameters; *input rate slack* (IRS), *output rate slack* (ORS) and *maximum output rate* (MOR). MOR is the inverse of compute time of the assigned edge to that

module, which is a constant value for a given configuration, i.e. the hardware platform, process and its parameters. These parameters are used to keep track of the rate at which a module can process input and generates output.

We know that the connections between modules can be of type C1M or CM1. Between any two consecutive layers, the modules which generate the output are called *Source Modules* (SM) and the downstream modules are called *Consumer Modules* (CM). For the case of C1M connectivity, the number of possible connections is equal to the number of fan in slots possible which is equal to the number of CMs. On the other hand, for CM1 connectivity the number of connections is equal to number of SMs. Algorithm 2 shows how the connections between processes are made. These connections are made between assigned modules of SRAG hence, generating an STG which will be used to run the application on the heterogeneous cluster. The algorithm uses a max priority queue called *sharedConVrtxQ*. This queue holds module vertices with a parameterized comparison policy to either compare IRS values of member modules or ORS.

In Algorithm 2, the *Clear existing topology mapping* step not only removes all module to module connections it initializes the IRS and ORS values to be equal to MOR. When a connection between two processes is made then slack updates are made as follows, where a subscript 'c' representing consumer module parameter and subscript 's' representing source module parameter;

if $IRS_c \geq ORS_s$ then

$$IRS_c = IRS_c - ORS_s \text{ and } ORS_s = 0.$$

On the contrary if $IRS_c < ORS_s$ then

$$ORS_s = ORS_s - IRS_c \text{ and } IRS_c = 0.$$

Algorithm 2. System Topology Creation

Input: *modVertices* v_m
Output: *system topology*
Clear existing topology mapping
srcLyr←*first element* ∈ l
for each *consLyr* | *consLyr* ← ∀ l *except first element* **do**
  if (*connectivity type***(***srcLyr, consLyr***)** ∈ *C1M connectivity* **then**
    set *sharedConVrtxQ compare policy* ←*ORS*
    insert $m \in v_m | m \in$ *srcLyr in sharedConVrtxQ*
    insert $m \in v_m | m \in$ *consLyr in array oneConVrtx*
    sort oneConVrtx in descending order of IRS
  else
    set *sharedConVrtxQ compare policy* ←*IRS*
    insert $m \in v_m | m \in$ *consLyr in sharedConVrtxQ*
    insert $m \in v_m | m \in$ *srcLyr in array oneConVrtx*
    sort oneConVrtx in descending order of ORS
  for each *module* m *of oneConVrtx* **do**
    if *sharedConVrtxQ not empty* **then**
      $scm \leftarrow$ *dequeue sharedConVrtxQ*
    if (*connectivity type***(***srcLyr, consLyr***)** ∈ *C1M connectivity* **then**
      make connection from scm *to* m
      if *ORS of* $scm \neq 0$ **then**
        insert scm *in sharedConVrtxQ*
    else
      make connection from m *to* scm
      if I*RS of* $scm \neq 0$ **then**
        insert scm *in sharedConVrtxQ*

These module node parameter updates help in keeping track of what is the available slack per module based on the connectivity. This information will be used in the throughput optimization algorithm to determine the bottleneck layer based on the number of modules in that layer and the associated input and output connectivity of modules.

Once the module parameters, IRS and ORS are computed based on the system topology, the *effective output rate* (EOR) for every assigned module in the topology and the *layer effective output rate* (LEOR) is computed for every layer in the topology. EOR for any given module m is computed as,

$$EOR_m = MOR_m - ORS_m$$

Finally, the LEOR a layer $k \in l$ is computed as,

$$LEOR_k = \sum_{m \in v_k} EOR_m$$

Where $v_k \in v_m$ for the given layer k.

2.4.3 THROUGHPUT OPTIMIZATION

The process of throughput optimization involves identifying bottlenecks and removing them layer after layer. If a MFS exists then, STG must be analyzed for bottlenecks. A bottleneck occurs if a layer with higher priority has higher throughput compared to its immediate layer with lower priority. The end-to-end throughput η_P of the pipeline and the bottleneck BL layer is

$$\eta_P = min_{k=1\ to\ l}(LEOR_k)$$

$$BL = k \in \eta_P$$

While comparing $LEOR$ values, the layer priority is used as a tie breaker. Therefore, if multiple layers have same output rate then the upstream layer is correctly identified as bottleneck layer, we call this operation as *getBottleneckLyr*. BL may not be the effective bottleneck layer when we are trying to scale the number of modules in a layer as we need to satisfy two kinds of constraints; slack and connectivity constraints at the bottleneck layer. Based on these constraints the effective bottleneck layer (*ebl*) is determined. The slack constraints (SLC) are defined as

$$SLC = \begin{cases} \sum_{m \in v_1} ORS_m = 0 \quad , & if\ BL = 1 \\ \sum_{m \in v_{BL+1}} IRS_m \neq 0, & if\ 1 < BL < l \end{cases}$$

If the bottleneck is at the first layer then the aggregate ORS must be saturated to warrant a scaling of this layer. On the other hand, when the SMs have saturated the input capacity of the CMs for $1 < BL < l$ while SMs belong to bottleneck layer then, scaling SMs will have no increase of overall throughput. Therefore, *ebl* would be the layer of CMs and this layer must be scaled. After *ebl* is determined the layers that must be scaled to resolve the bottleneck is determined based on the C1M and CM1 constraints. A list of these layers is called as affected layers (*AL*). The expression for connectivity constraints (COC) used in the algorithm to determine *AL* which is based on based on C1M and CM1 is defined as

$$COC = \begin{cases} |m_1| + 1 \leq |m_2|, & if\ BL = 1 \\ |m_2| + 1 \geq |m_1|, & if\ BL = 2 \\ |m_c| + 1 \leq |m_s|, & if\ 2 < BL < l \end{cases}$$

The candidate layer for scaling is the layer which satisfies SLC and COC constraints is the effective bottleneck layer. Algorithm 3 shows the details of steps involved in throughput optimization. In the algorithm while cloning a vertex we clone its request and assigned edges only as these are still viable options for mapping.

Algorithm 3. Throughput Optimization

Input: *modVertices* v_m
Output: *optimized SRAG and corrosponding STG*
create STG
done ← *false*
while ¬*done* **do**
    ebl ← *getBottleneckLyr*
    for each *lyr* ← *btlnkLyr to* l **do**
        ebl ← *lyr*
        if *SLC for lyr is satisfied* **then**
            break
    if *ebl* = l **then**
        break

    clear *AL*
    if $ebl = 1$ **then**
        insert ebl *to AL*
        if COC *is not satisfied for* ebl **then**
            $ebl \leftarrow ebl + 1$
    insert lyr to AL| lyr ∈ *ebl to 2, until COC is satisfied*

    for each *lyr* ∈ *AL* **do**
        e_r ← *edges of type Request or Assign* $\forall v_m \in lyr$
        if $|e_r| = 0$ **then**
            done ← *true*
            break
        else
            sort e_r *in ascinding order of* τ
            madeAssignment ← *false*
            for each *edge* $e \in e_r$ **do**
                if *assignment of* e *is possible* **then**
                    v' ←*clone process vertex of* e
                    set e' *type* ← *Assign | clone of* $e, e' \in v'$
                    create STG
                    madeAssignment ← *true*
                    break
                else
                    set e *type* ← *Invalid*
            if *madeAssignment = false* **then**
                done ← *true*
                break

If the last layer is the bottleneck layer then and if further scaling is required the (talk about

extension of this work where this problem is treated recursively in a bottom up approach for a larger pipeline of SLP pipelines.)

2.5 Structure Based Runtime Scheduling

The number of modules in the STG varies based on the available cluster resources and the connectivity between the modules is not pre-determined at application design time though the connectivity pattern is fixed. Due to these reasons, the STG can vary for the same application running on the same cluster for different runs. This poses a challenge for scheduling tasks for such a non-deterministic setup. The connections in the STG are point-to-point therefore special care must be given to ensure deadlocks at the system level don't occur due to improperly scheduled workloads. We address this problem by employing novel structure based scheduling which is capable of both dynamic load balancing and congestion control in a de-centralized way.

SLP works on fork and converge methodology. This has an advantage that a high-level scheduler for the overall pipeline is required only in the first layer. This scheduler has the knowledge of the topology graph and makes decisions such that all dependent tasks converge to the same downstream module for task resolution so, we call this *Structure Based Scheduler* (SBS). SBS analyzes STG from bottom-to-top to determine the connectivity and creates sets of hierarchical groupings of modules present in the second layer as scheduling is done only at the first layer. The levels in this hierarchy is defined as $1 \leq L \leq l - 3$. SBS is present in every module of first layer and while analyzing the STG it looks at paths that are only visible to it. Each level of the hierarch represents a reduction (converge) operation of tasks in its corresponding layer. The different groups of a level imply number of independent parallel tasks that can be scheduled for the layer corresponding to that grouping level. Each group is a contains layer 2

modules representing parallel paths for scheduling any given task. The last layer with only one module just collects all the results. Hence, we require $l-3$ levels to determine the appropriate scheduling path.

We create this grouping of modules for every layer $3 \leq k \leq l-1$. A group $g_{Lx} \in S_L$ where $1 \leq x \leq |m_k|, L = l-k$ for the corresponding layer k, resulting in $|m_k|$ groups belonging to level L in every set S_L. These groups are hierarchical where a group g_{Lx} contains sub groups from level 1 to $L-1$. We annotate these groups with level numbers and module number x as subscripts $\{\}_{Lx}$ to keep track of hierarchy. Each group g_{Lx} contains only the layer 2 modules which are visible from module m_{kx} and SBS, we call this the *Visibility Condition* (VC). This grouping is determined as k varies from $l-1$ down to 3. The starting case $k = l-1$ implies level-1 set S_1 has all the layer 2 modules visible from m_{l1} and SBS with $|m_{l-1}|$ groups, with each group containing layer 2 modules which satisfy VC for every module m_{l-1x}. The grouping for level 1 and arbitrary level L are;

$$S_1 = \left\{\{m_{21}, m_{22}, \dots\}_{11}, \{\dots\}_{12}, \dots, \{\dots, m_{2|m_2|}\}_{1|m_{l-1}|}\right\}$$

$$S_L = \left\{\{\dots\{\dots\{m_{21}, m_{22}, \dots\}_{L1}, \dots\}_{L-11}, \dots\}_{11}, \{\dots\}_{12}, \dots, \left\{\dots, \left\{\dots, \{\dots, m_{2|m_2|}\}_{L|m_{l-k}|}\right\}_{L-11}\right\}_{1|m_{l-1}|}\right\}$$

In this way, we determine all the level based grouping. For example, let's consider the STG shown in Fig. 7. After analyzing the topology from the perspective of module m_{11} SBS, the level based groupings are shown below:

$$S_1 = \{\{m_{21}, m_{22}, m_{23}\}_{11}, \{m_{24}\}_{12}\}$$

$$S_2 = \{\{\{m_{21}, m_{22}\}_{21}, \{m_{23}\}_{22}\}_{11}, \{\{m_{24}\}_{23}\}_{12}\}$$

S_{l-3} is a super set containing the grouping information of all the levels. SBS schedules the tasks based on the level based grouping structure of S_{l-3}. Level 1 grouping represents the

scheduling options for the most dependent task in the task dependency graph t_{l-1i}, where i is an index of task belonging to layer $l-1$. The modules present in one such group are the possible scheduling options for dependent tasks of t_{l-1i}. SBS picks one group based on a scheduling policy (for example round robin) and picks one of the sub-groups from the selected group to determine the scheduling options for the next set of dependent tasks of t_{l-1i}. In this way, SBS walk through the grouping hierarchy till the independent base tasks are analyzed, which corresponds to grouping for level $L = l - 3$. There are no sub groups at this level. All the modules in this group can be picked to schedule the base tasks. We call this group as a pool $P \in g_{l-3}$. SBS now schedules the base tasks to these pools. Any layer 2 module associated with that pool gets a task whenever it is ready to process the next one in a dynamic manner. This mechanism allows for dynamic load balancing as modules ready for computation get jobs asynchronously. If there is a congestion at a module i.e. it is taking too long to finish a job, then it doesn't get new jobs as its input buffer would have filled up in the meantime. Due to this the immediate upstream module cannot forward its results thereby filling up its output buffer. Hence stalling the immediate upstream modules micro pipeline. This effect cascades upstream asynchronously. In this way SBS guaranties that the right path is selected to schedule the base tasks along layer 2 modules which eventually leads to dependent tasks of same hierarchy converging to the same downstream module asynchronously while dynamically selecting the fastest computation path down the topology graph.

2.6 Analytical Simulation Results

The SLP framework is implemented using C++ along with ELPC model for comparison. We generate a set of experiments to test and compare the analytical models. These experiments are

generated by randomly varying parameters such as the number of nodes including the variety of heterogeneous computing nodes, the process parameters viz; number of threads, number of accelerators needed with their proportionate runtime and memory bandwidth requirements. The simulation results are shown in TABLE I.

TABLE I. Simulation results comparison

Sl. No.	Num. Layers	Nodes in cluster	SLP			ELPC		
			Modules assigned	Nodes used	Throughput	Modules assigned	Nodes used	Throughput
1	10	12	12	6	0.4292			
2	13	10	13	7	0.3165			
3	16	13	29	12	0.7752			
4	17	15	34	13	0.8522			
5	24	25	46	21	0.7673			
6	25	20	30	15	0.4673			
7	30	27	77	27	1.0817			
8	33	29	65	26	1.1539			
9	6	9	21	9	1.7667	6	6	0.3846
10	8	12	19	9	1.3553	8	8	0.4184
11	9	18	53	16	2.9495	9	9	0.4630
12	10	15	40	12	1.5628	10	10	0.3584
13	12	13	29	12	0.9738	12	12	0.3401
14	15	18	42	14	1.0728	15	15	0.3448
15	16	27	53	24	1.4741	16	16	0.5435
16	18	20	51	19	1.2207	18	18	0.3135
17	20	23	64	22	1.5433	20	20	0.5917
18	21	25	79	25	1.6177	21	21	0.3086
19	22	23	61	19	1.2219	22	22	0.2381
20	29	29	83	28	1.1220	29	29	0.2421

We can see that SLP is flexible enough to map a pipeline on highly constrained resources i.e. when the number of nodes is less than the number of simultaneous modules that need to be mapped. We can also see that the proposed model outperforms ELPC in terms of end-to-end throughput even when the number of modules is equal to number of available nodes, which is the best scenario for ELPC as it cannot scale. Experiments 1 to 8 couldn't produce any solution using ELPC algorithm for the following two reasons. Reason 1; the number of nodes available in cluster is less than the number of layers required. Reason2; the experiments run on a heterogenous cluster and a module can run on only limited number of nodes hence causing resource conflicts which cannot be resolved using ELPC algorithm as it doesn't support

simultaneous resource sharing. Experiments 1 and 5 failed to produce an ELPC solution due to reason 2. Experiments 9 to 20 show the throughput achieved due to scaling introduced by SLP over ELPC.

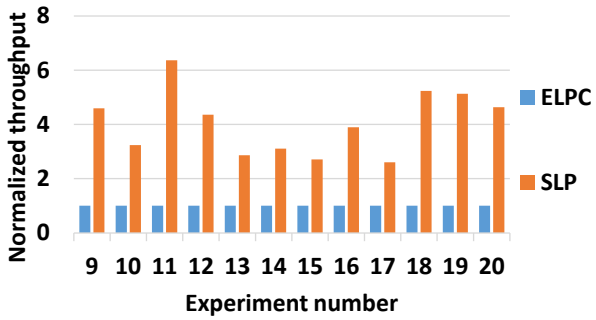


Fig. 9. Throughput gain over ELPC

Fig. 9 shows the plot for performance gain normalized to ELPC throughput. We can clearly see that SLP outperforms ELPC. Fig. 10 plots the ratio of modules per node for both models to show the efficiency of resource utilization.

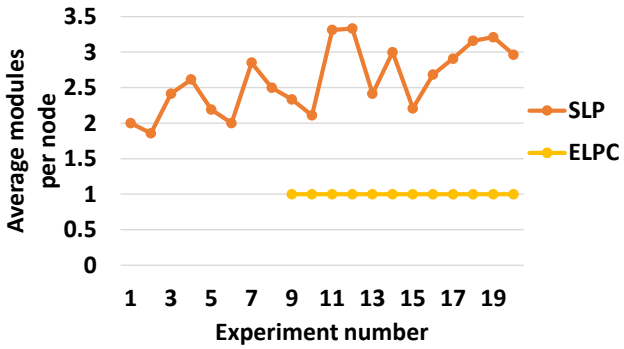


Fig. 10. Resource utilization efficiency

Experiment 9 undergoes 13 evolutions of STG during the throughput optimization phase resulting in a throughput of 1.7667 unit work per second. The intermediate STG results for this experiment is shown in Fig. 11 along with its throughput. Each process can run with different configurations hence, ELPC algorithm is provided with configurations with best possible run time so that it can achieve maximum throughput.

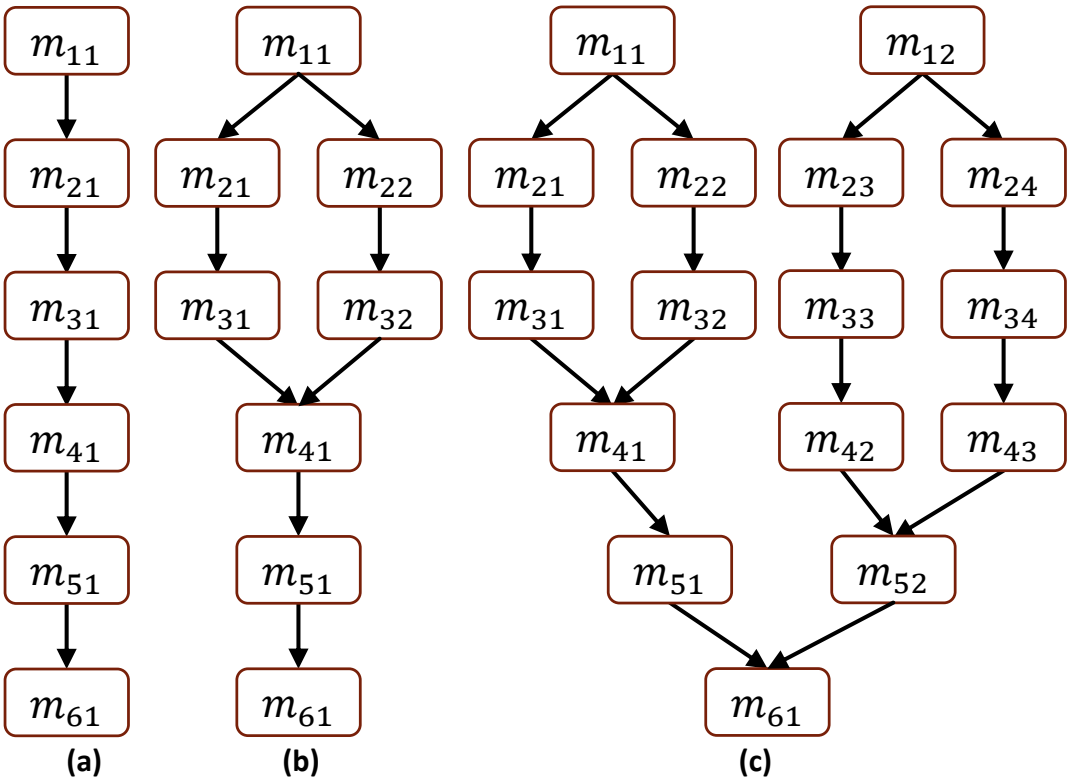


Fig. 11. Intermediate STGs for experiment 9 (a) MFS, $\eta_P = 0.5988$ (b) Evolution 1, $\eta_P = 0.7042$ (c) Evolution 8, $\eta_P = 1.5375$

3 SLP VALIDATION

To validate the proposed SLP framework with a real-world application we pick a neuromorphic application called Intelligent Text Recognition System (ITRS). We modularize it and implement it on a heterogeneous cluster and demonstrate the ability of the application to scale with the available resources. We also compare the SLP and ELPC model.

3.1 INTELLIGENT TEXT RECOGNITION SYSTEM

The neuromorphic model adopted by the ITRS software is mainly built based on the Brain-State-in-a-Box (BSB) attractor model [32] [33] [34] and the Cogent Confabulation model [35]. The BSB models provide matching patterns for each character image. The cogent confabulation algorithms combine information from the BSB model to form more complex objects such as words or sentences. During this procedure, it suppresses the inputs that do not have strong association with others and enhances the remaining inputs. In other words, the confabulation model eliminates those BSB results that do not form meaningful words and sentences. Therefore,

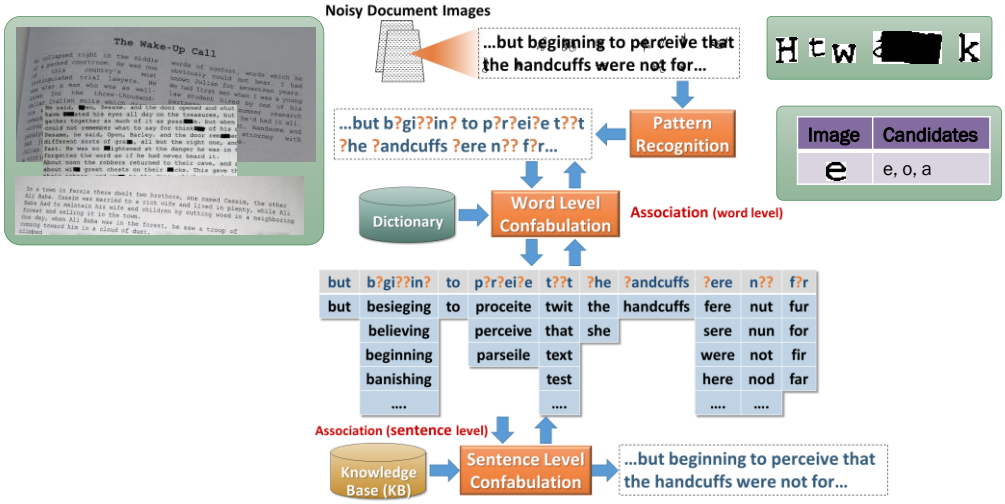


Fig. 12. ITRS cognitive model

ITRS is capable of extracting meaningful text from noisy and occluded document images. The salient feature of ITRS is that it provides contextually correct sentence reconstruction even if there are illegible characters or words in the document image [2]. The cognitive model of ITRS is illustrated by an example in Fig. 12.

Given a noisy document image, the BSB provides pattern-matching candidates for each character image using best effort. Each question mark in the figure represents all 26 possible alphabets, numbers and commonly used symbols. The word confabulation layer builds word candidates while filtering out any meaningless words and the sentence confabulation layer selects the words that form the most meaningful sentence. It is easy to see that, for each sentence, one sentence confabulation task and multiple word confabulation tasks must be executed, along with even more number of BSB pattern matching tasks. The computation tasks within the same level are independent to each other and hence can be implemented in parallel. Based on the discussion above there are different distinct stages in the pipeline with different compute requirements. Since image processing applies different algorithms for processing different regions of the noisy image, it is a thread intensive task where each thread performs small but distinct computation. Therefore, each type of computation gets a thread pool for efficient asynchronous computation. BSB is an attractor model of auto associative memory, it is compute intensive and best suited for high performance accelerators. Word and sentence confabulation tasks perform sparse computation and have intensive random memory access. Such variety in workload characteristic and data dependencies is common in full-scale neuromorphic applications.

The ITRS is modularized based on its functional stages as shown in Fig. 13. This modularization and parallelization method enable us to separate individual modules and run them on most appropriate hardware platform that fits their computation requirement. Hence it is

capable of effectively utilizing resources in a heterogeneous cluster.

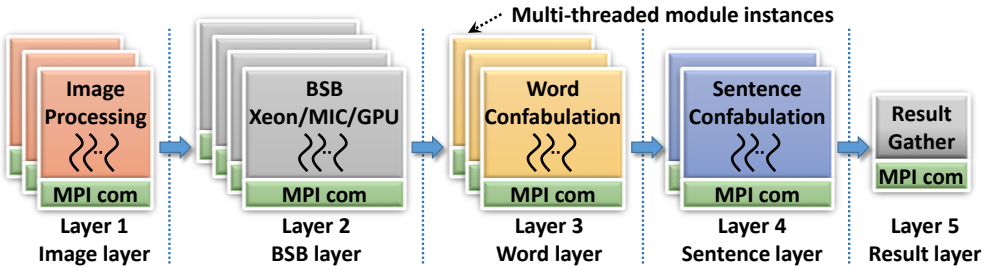


Fig. 13. ITRS pipeline

We adopt the work presented in [36], which talks in more detail about the communication module but it doesn't provide any analytical model for the pipeline or perform any resource mapping and optimization. The MPI communicator which runs using one thread is implemented as a reusable library. It is designed to interface in a thread safe way with all modules in the pipeline. It is a key enabler for reliable and scalable implementation. Each module can only have one such communicator as it is able to initiate and keep track of multiple non-blocking communication with any MPI rank simultaneously. Apart from data messages which are non-blocking, it also supports sending and receiving status messages to and from multiple ranks to facilitate inter-module synchronization. All status messages are one byte integer messages and data messages are given size character messages. Status receive is non-blocking but; blocking MPI communication is used for status transmit. In this way we can guarantee that the status is sent in order with respect to that modules data messages. In this work, the focus of our attention is the analytical model and its validation, therefor we don't go into implementation details of the communicator and the scheduler.

3.2 UNIFORM INTER-MODULE COMMUNICATION

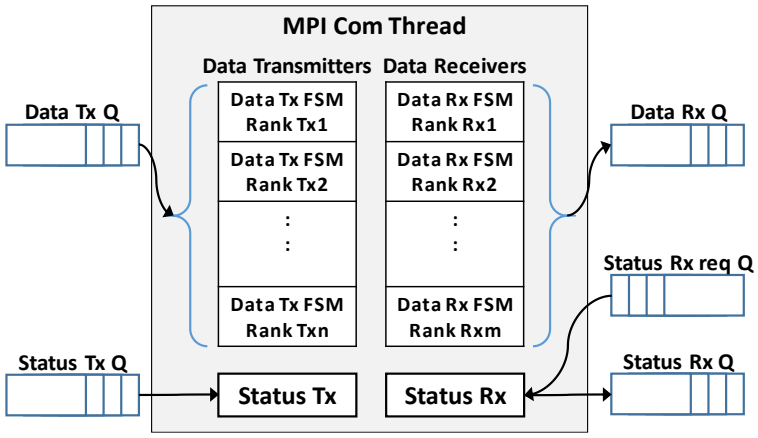


Fig. 14. MPI communication sub-module architecture

The MPI communication sub-module (MCSM) which is a reusable library, is designed to interface in a thread safe way with all modules in the pipeline. It is a key enabler for reliable and scalable implementation. MCSM is always attached to a software module, which is also referred as its *parent module*. Each functional module can only have one MCSM.

MCSM contains only one thread and all messages are funneled through this thread. It can send and receive data messages from multiple ranks simultaneously. Apart from data messages, it also supports sending and receiving status messages to and from multiple ranks to facilitate inter-module synchronization. All status messages are one byte integer messages and data messages are given size character messages. While non-blocking MPI communication is used for data transmission and status receiving; blocking MPI communication is used for status transmit. In this way we can guarantee that the status is sent in order.

The architecture of this sub-module is shown in Fig. 14 MCSM manages five thread safe blocking queues. As shown in the figure, they are for data send (Data Tx Q), data receive (Data Rx Q), status send (Status Tx Q) and status receive (Status Rx req Q and Status Rx Q). The parent module is the producer for Data Tx Q, Status Tx Q, and status Rx req Q. It is the consumer

of Data Rx Q and Status Rx Q. MCSM doesn't block on any of these queues to maintain reliable, always open communication. However, depending on the computing requirements of the threads in the parent module these queues can be used in a thread non-blocking or thread blocking manner based on the full and empty flags of the queues. As we mentioned before, each received data message specifies a job to be processed. If the parent module consists of multiple threads, all threads fetch jobs from Data Rx Q whenever they are free.

MCSM maintains two types of data objects; Data Transmitters and Data Receivers. It creates one data transmitter or data receiver object for each MPI rank it needs to communicate with. Each of these objects contains a state machine to manage the communication with that particular rank.

The data messages generated by the parent module consist of destination rank information which are forwarded to MCSM through Data Tx Q. Each Data Transmitter object is interfaced with a local queue. These queues are stored in a map container with the destination rank of the associated object as the key. Whenever there is data available on the Data Tx Q it is copied on to appropriate local queue with the same key. Transmitter pools are created by sharing the local queues among select data transmitter objects. The data messages in the local queue can be sent to any rank in that pool as long as it is free to receive. Therefore, these transmitter pools enable load balancing in a distributed way at module level among the ranks in the pool. The data receiver objects en-queue all received data messages to Data Rx Q. It is the responsibility of the consumer of this queue to de-queue data messages from them.

MCSM also maintains one object for the status transmitter and receiver, and it is shared among all ranks. As we mentioned before, blocking MPI communication is performed for status information. The status communication is only used during initialization and termination.

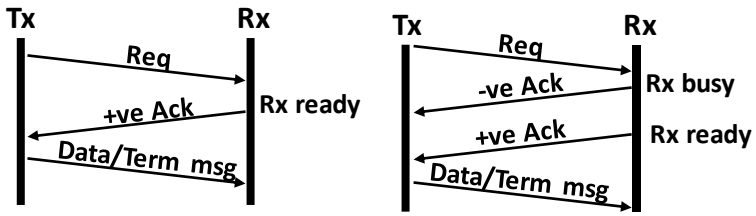
The Status Tx Q receives status messages from the parent module along with the destination

rank to MCSM. The Status Tx object sends one message at a time by initiating a blocking MPI send. To request the status message from a particular module, the parent module sends the rank of the target module to Status Rx Req Q. This initiates a non-blocking MPI receive. Another status request is not processed until current one is complete. The received status message is en-queued to Status Rx Q. We make the receive operation non-blocking and give power to the parent module to decide whether to block on an empty Status Rx Q or not.

The MPI communication thread keeps on polling the TX Q and Status Req Q, hence it is always busy from creation to termination. The thread round robins on four functions: data receive, status receive, data transmit and status transmit. The detailed communication protocol is discussed next.

3.3 COMMUNICATION PROTOCOL

All communication is point-to-point and follow a flow control based communication protocol. The receiver has a capacity limit, which is the maximum length of Data Rx Q. If there is room for data in the queue then the receiver is inferred to be in ready state else it is inferred to be in busy state. In this protocol the transmitter sends a request message to the receiver. Based on the state of the receiver there can be two cases as shown in Fig. 15. If the receiver is ready, then it sends a positive acknowledgement to the transmitter, otherwise it sends a negative acknowledgement . Once the receiver becomes ready, it will again send a positive acknowledgement message. The transmitter sends data only after receiving a positive acknowledgement. If a negative acknowledge is received, it will yield its turn to other transmitter objects, in this way load balance is achieved. These protocols are implemented as state machines on transmitter side and receiver side. The description of the state machines is given below.



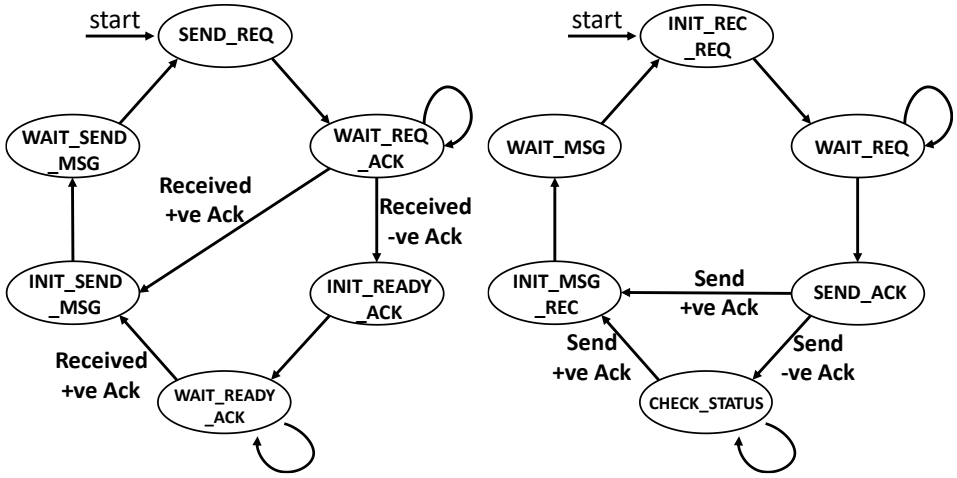
(a) (b)

Fig. 15. Flow Control Protocol. (a) Receiver is ready (b) receiver is busy

The Transmitter State Machine (TSM) has six states as shown in Fig. 15 (a). The TSM starts in send request state (SEND_REQ) where, a request message is sent to the receiver and the state transitions to wait request acknowledgement (WAIT_REQ_ACK) state. TSM now waits for acknowledgement message from the receiver. If it receives a positive acknowledgement, then it transitions to initiate send message (INIT_SEND_MSG) state else it transitions to initiate ready acknowledgement state (INIT_READY_ACK) where a non-blocking MPI receive is called and the state transitions to wait ready acknowledgement (WAIT_READY_ACK). After receiving an acknowledgement, the TSM transitions to INIT_SEND_MSG where a non-blocking MPI send is called and the state transitions to wait send message (WAIT_SEND_MSG) where it waits till the MPI message is sent. Then it transitions to SEND_REQ and the process starts all over.

The receiver state machine (RSM) also has 6 states as shown in Fig. 15 (b). The RSM starts in initiate receive request state (INIT_REC_REQ) where, a MPI non-blocking receive is called and the state transitions to wait request (WAIT_REQ) state. RSM now waits for a request from the transmitter. Upon receiving the request, it transitions to send acknowledgement (SEND_ACK) state. In SEND_ACK state the RSM checks if Data Rx Q is full and sends a positive acknowledgement if there is room for data then the state transitions to initiate message receive (INIT_MSG_REC) state. Otherwise, if the queue is full then a negative acknowledgement is sent and the RSM transitions to check status (CHECK_STATUS) state. In this state the RSM

monitors the Data Rx Q size, as soon as the parent module de-queues a data message the RSM sends a positive acknowledgement and the state transitions to INIT_MSG_REC state. In INIT_MSG_REC state a non-blocking MPI receive is called and state transitions to wait message (WAIT_MSG) where RSM waits till message is received. After receiving the message, state transitions to INIT_REC_REQ and the process starts over.



(a) Transmitter State Machine (b) Receiver State Machine

Fig. 16. Communication protocol state machines

The TSM (RSM) will make one transition according to the diagram when the MCSM enters the data transmit (data receive) function in the round robin process.

3.4 ITRS MICRO PIPELINES

Each module in the ITRS pipeline can have different internal architecture based on its implementation platform. In this section, we discuss only the high-level design details for different stages of ITRS processing without going into details of algorithm design and optimizations for different hardware platforms.

3.4.1 IMAGE PROCESSING LAYER

The image processing layer works on extracting individual characters from noisy images. This layer works on partial regions of the image hence not suitable for vectorization. Therefor this layer can be parallelized using thread pools dedicated to handle different tasks. The preferred hardware platform to run image processing module would be a Xeon processor platform as it supports multiple threads and has ideal resources to handle small workloads. For effective character extraction various settings are provided to tune image processing which can vary over different runs based on the type of images and the noise present in them. We make very few assumptions on the input quality of the image, hence a robust image processing is implemented. To achieve high throughput this module is implemented as a pipeline with 6 stages as shown in Fig. 17. Every stage is designed to run as an independent thread. To improve performance each stage is a configurable thread pool, except for the first stage which always runs with a single thread. The pool size for each stage is configured so that there is optimal load balancing among the stages with minimum thread idle time. For efficient implementation, OpenCV is used to develop this algorithm.

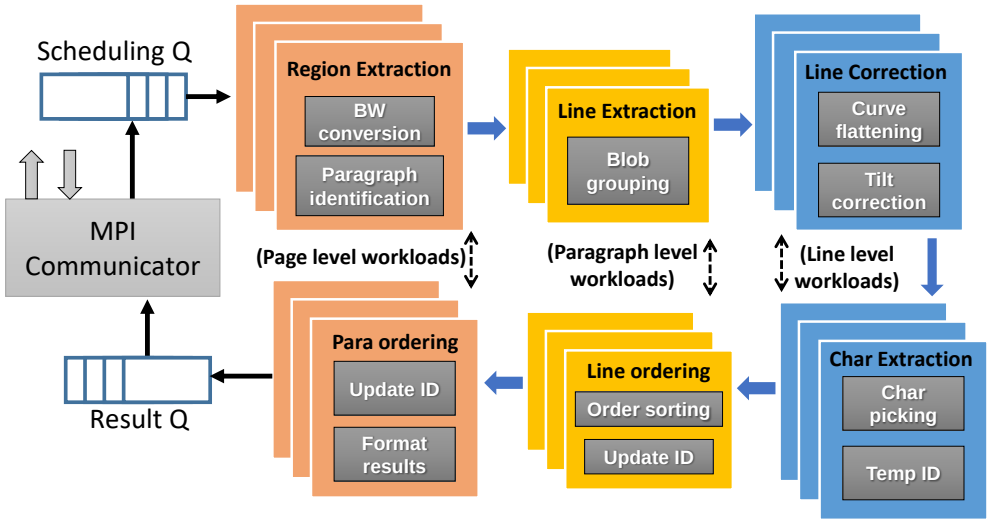


Fig. 17. Image processing pipeline

The interface between the pipeline stages is a blocking queue. This method of communication is thread safe and also optimizes resource utilization as a thread is blocked till data is available. The functionality of each stage is described below:

3.4.1.1 REGION EXTRACTION

This is the first stage in image processing and operates at the page level. In this stage multiple pages processed in parallel and are broken down to paragraphs. It performs basic image filtering to reduce noise in the images and to prep them for Black and White conversion. A 2D gradient of image is computed to extract blob boundaries and removes any illumination gradient usually present if the document image is captured using camera with flash. Every blob is picked up one at a time and converted to black and white using Otsu's method. High fidelity conversion is possible due to dynamic threshold computation for every blob, rather than one threshold for entire image which often leads to poor results.

Morphological erosion is performed on the white background of the image with a rectangular structuring element. The size of the rectangle is chosen such that the height is larger than the text line spacing and width is lower than the column gap in the document image. This operation results in a hole for each paragraph location in the background. Using this as a mask the paragraphs are extracted and labeled. Each paragraph is a workload for the next stage in the pipeline. These workloads can now be processed in parallel to extract text lines from the paragraphs.

3.4.1.2 LINE EXTRACTION

This stage operates at paragraph level and the text lines from paragraph are extracted. To begin with, connected components in the paragraph are identified and labeled. The bounding boxes for these blobs are also determined. Now the paragraph image is scanned from top to down

and left to right order. Once a connected component is encountered, it is labeled as an initial blob for the line. The region to the right of the initial blob is scanned to search for a neighboring blobs. The width of the search region is large enough to at least include a text space and few characters of the next word. The height is based on the height of the bounding box of the previous character, in this case the initial blob. Few extra buffer rows are included on the top and bottom of the search region to include blobs from disjoint characters like 'i', 'j', '%' etc. Any blob in this search space is picked up and marked as current line. The bounding box for the disjoint character is updated to reflect the actual character height. This new found character/blob height and position is used as starting point for next search. The searching and labeling process continues until the page boundary is reached or until the search space is empty. At his point the same process is repeated starting from the initial blob but in left direction until the page boundary is reached or until the search space turns out empty. In this way a text line is labeled and extracted from the paragraph image. Each extracted line is treated as workload for the Line Correction stage.

3.4.1.3 LINE CORRECTION

This stage operates at line level. In this stage any deformations due to warping and rotation are corrected. The mid points of all the blobs in the line are used to build a polynomial regression model. The degree of the polynomial is chosen to be 3 as it is sufficient to model the nonlinearities present in the above mentioned distortions. Once the polynomial is computed, the values of the coefficients are analyzed and the degree of the polynomial is reduced to avoid over estimation. This ensures a smooth and proper polynomial fitting model. After the model is computed the polynomial curve is used as a reference line. This reference line is scanned from top to bottom of the extracted line. All pixels along the scan curve are saved as a flat line thus eliminating warping. At this stage the characters will be tilted if there was rotation. The y-offset of

the starting point and the end point of the reference line is computed. This offset is the indication of angle of rotation as shown in Fig. 18.

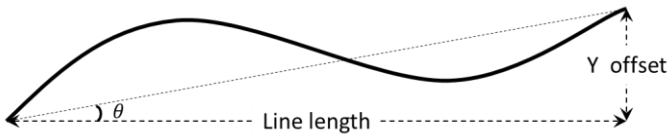


Fig. 18. Angle computation

The angle θ is computed using the relation $\theta = \tan^{-1}\left(\frac{y\ offset}{Line\ length}\right)$. Fig. 19 represents the flattened scan lines with the gray region representing the actual text region and solid horizontal lines representing each pixel row. The tilt can now be corrected by offsetting the starting points (P1, P2 … Pn) of these lines. The angle ϕ is computed using the relation $\phi = \frac{\pi}{2} - \theta$ and the shift distance for each point is given by the equation $x_{offset} = \frac{row}{\tan(\phi)}$. This operation corrects the tilt introduced due to rotation. If the line is tilted with an angle "$-\theta$" then the points are shifted in the opposite direction. The corrected lines forms the workload for character extraction stage.

Fig. 19. Tilt correction

3.4.1.4 Character Extraction

This stage operates at the line level. Here the characters are extracted and scaled resulting in removal of perspective distortion. The first step in this stage is to morphologically group connected component blobs into individual characters. Now the blobs are sorted in column order and each blob is treated as individual character. These are extracted and labeled with page ID, paragraph ID, sentence ID, word ID and character ID. The sentence ID is incremented when a ','

or '.' character image is encountered. These two characters are identified based on the dimensions and position statistics of the character image in the line. The word ID is incremented when a space width threshold is met between the characters. The characters are labeled as though they are the first line. As the only information available at this stage is about current text line. This stage also splits any connected characters or occlusions based on the configuration specified. Once the characters are labeled they are scaled to either 15x15 or 30x30 resolution. This scaling to a fixed size eliminates perspective distortions introduced in camera captured images.

Each paragraph workload from Region Extraction stage is assigned a thread safe counter called Line Counter. After character extraction is finished it increments this counter. Hence this counter keeps track of the number of lines processed in the paragraph. If all lines are processed then the parent paragraph of the current line is scheduled to Line Ordering stage.

3.4.1.5 LINE ORDERING

This stage operates at the paragraph level. This stage determines the correct order of text lines in the paragraph region. To begin with, it computes the average angle of tilt of all the lines in the paragraph. This information gives the angle of rotation of the image (here paragraph). The starting point of each text line is rotated back by the angle computed. The rotation is performed using the equation

$$\begin{bmatrix} RP_{x1} RP_{x1} & & RP_{xn} \\ RP_{y1} RP_{y2} & \cdots & RP_{yn} \end{bmatrix} = H * \begin{bmatrix} LP_{x1} LP_{x1} & & LP_{xn} \\ LP_{y1} LP_{y2} & \cdots & LP_{yn} \end{bmatrix}$$

Where H = is the rotation matrix $\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}, \theta = \frac{\pi}{2} - average\ angle, LP$ = Line point RP = Rotated point. By doing this the y co-ordinates of these points correctly represent the line order. This process has now sorted the lines in order. Using this information the sentence ID and word ID of all the extracted characters in this paragraph are updated. The result at the end of

this stage reflects correct sentence ID, word ID and character ID at the paragraph level.

Each page workload generated for Region Extraction stage is assigned a thread safe counter called Para Counter. After line ordering is finished it increments this counter. Hence this counter keeps track of the number of paragraphs processed in the page. If all paragraphs are processed then the parent page of the current paragraph is scheduled to Para Ordering stage.

3.4.1.6 PARA ORDERING

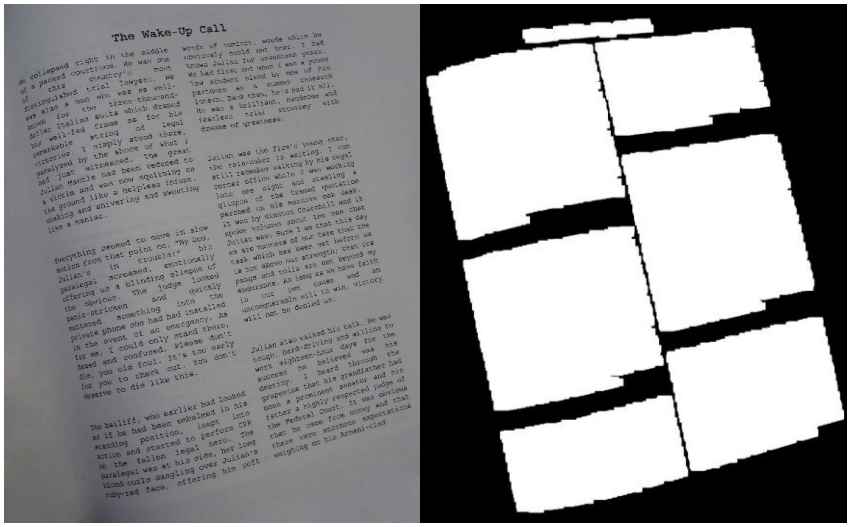
This stage operates at the page level. In this stage final ordering and result generation is performed. All the paragraph starting points are rotated similar to Line Ordering stage. Based on the average angles of the paragraphs. This point positions are used to group the paragraphs into text columns and the paragraph order in each column.

The points are sorted based on column position. The first paragraph starting position is used as reference and any paragraph whose starting position falls within a threshold of column positions is grouped as one column. This process is repeated till all the paragraphs are grouped into columns. After this grouping each column group is sorted row wise to get the proper order. Based on this order the sentence ID and word ID of all the extracted characters are updated to reflect the correct order. After ordering is completed the results are scheduled to the pattern matching layer. The intermediate results of image processing are shown in Fig. 20.

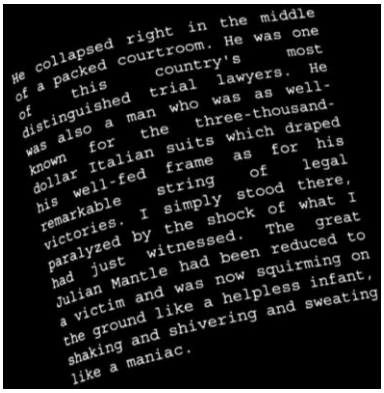
3.4.2 PATTERN MATCHING LAYER

The pattern recognition layer is based on BSB attractor model. In this section, we describe the "racing" mechanism that we use to implement the multi-answer character recognition process.

Let S denote the set of characters that we want to recognize. Without loss of generality, assume the size of S is 52, which is the number of upper and lower case characters in the English



(a) Original page (b) Page image after morphological erosion



(c) Intermediate image after region extraction

(d) Intermediate image after line extraction

(e) Intermediate image after line correction

Fig. 20. Intermediate results during image processing

alphabet. We also assume that for each character, there are M typical variations in terms of different fonts, styles and sizes. In terms of pattern recognition, there is a total of 52*M patterns to remember during training and to recognize during recall. Sometimes, the number of patterns is higher if digits and commonly used symbols are considered.

One 256-dimensional BSB model is trained for each character in S. Therefore, there will be a set of 52 BSB models. Each BSB model is trained for all variations of a character. The multi-answer implementation utilizes the BSB model's convergence speed to represent the similarity between an input image and the stored pattern. An input image is compared against each one of

the 52 BSB models; therefore, it triggers 52 recall processes. The number of iterations that each recall process takes to converge is recorded. Then we pick up to K "closest" candidates to work with high-level language models to determine the final output. Fig. 21 gives an example of how the racing mechanism works. Different hardware architecture specific optimizations can be made for this module, one such optimization is discussed in [37].

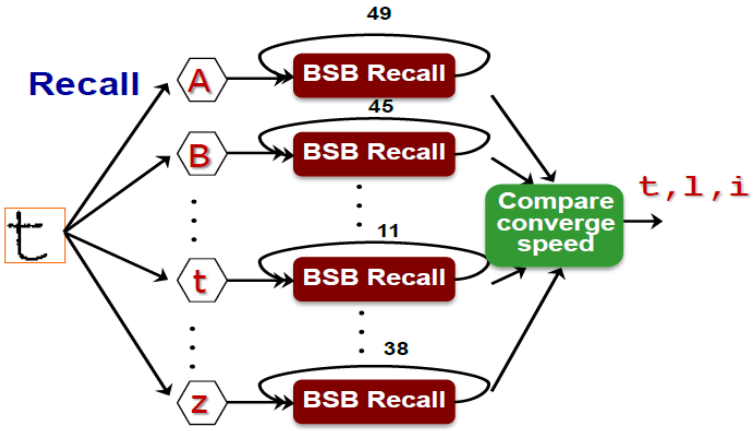


Fig. 21. Candidates are selected based on speed of convergence

3.4.3 Word Confabulation Layer

This module interfaces between BSB and Sentence confabulation. Its role is to collect ambiguous character inputs from BSB layer and generate valid combinations to form meaningful words. The architecture of word confab is shown in Fig. 22. The MPI communication module is integrated with word confabulation thread for MPI communication from-BSB and to-Sentence confabulation modules. All inter thread communication is achieved through the thread safe blocking queue. A word scheduler thread communicates with the MPI Communicator, and groups the results into words. These words are scheduled to Confabulation threads using the Scheduling queue. The number of confabulation threads created is user specified. These threads pick tasks from the Scheduling Q whenever they are free. If data is not available then that

particular thread is blocked till data is available as the Scheduling Q is a blocking queue. The results of word confabulation are converted to character messages and en-queued on to a common Result Q which are intern sent to sentence confabulation.

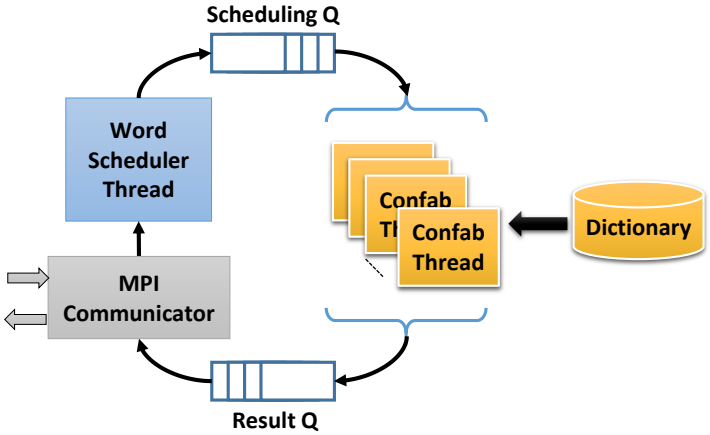


Fig. 22. Word confabulation architecture

The scheduler thread uses a map data structure to keep track of what all characters it has received which is called as word pool. From every received character a unique word level string key is built. The key is concatenation of [Page-ID]_[Sentence-ID]_[Word-ID]. The word pool consists of word lexicons as its data members. If a key is used for the first time then a new word lexicon is created and the received character candidates are added to it. If a word lexicon already exists then these letter candidates are added to the existing one. After adding the candidates it is checked if all are received for that word lexicon. After receiving all, the word lexicon is en-queued on the Scheduling Q and the map entry is erased.

The confabulation threads use the ambiguous letter candidates and creates valid word combinations. To accomplish this a dictionary database is loaded as trie data structure during initialization of module. This trie is shared between all threads which read this data structure to validate word combinations. An example of trie data structure is shown in Fig. 23.

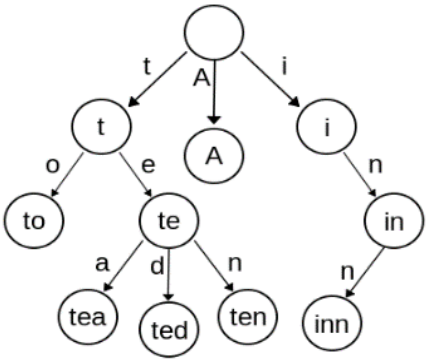


Fig. 23. Trie data structure

Each node holds following members a) Address to parent, b) Address of children, c) Content and, d) Word Marker. For example let's consider a word "dog". Its candidates for each letter position are [d o b] [o a g] [g a y]. Word confab will traverse through the trie using these candidates to search for the valid words present in the trie. In the above example, first it will find if 'd' exists as 1st letter, if it does, then it will go down that branch and look for 'o' as 2nd letter and so on until it hits the last letter which will be 'g' in this case. Then it will see if the work marker is 1 for that, it mean it's a valid word, otherwise it will go up 1 branch and check for the next letter. Once one branch is complete, it will move to the next branch until it has covered all the candidates. The valid words will be pushed onto a stack. In this example, the valid words would be: dog, day, boy and bag.

Since the letters candidates were passed with their relative confidence level, the confidence level for each word will be the product of the confidence levels of letters it contains. These results are converted to packed message. If the size of this message is large than the configured communication buffer, then they are split into multiple messages. These are now sent to sentence confabulation module.

3.4.4 SENTENCE CONFABULATION LAYER

Sentence level confabulation model defines three levels of lexicons. The first and second level lexicons represent single words and pairs of adjacent words; while the third level of lexicons represent the *parts-of-speech* (POS) tags of the corresponding word. During recall, those word and word-pair symbols corresponding to the outputs from word level confabulation are set as active, and all POS tag symbols are also set as active. If a lexicon has more than one active symbol, it is said to have ambiguity. The goal of sentence confabulation is to resolve the ambiguity iteratively through a recall procedure similar to belief propagation and finally form a meaningful sentence. The general confabulation recall algorithm can is described as follows in Algorithm 4.

As Algorithm 4 shows, for each lexicon that has multiple symbols activated, we calculate the *excitation level* of each activated symbol. The *N* highest excited symbols in this lexicon are kept active. These symbols will further excite the symbols in other ambiguous lexicons. This procedure will continue until the activated symbols in all lexicons do not change anymore. If the convergence cannot be reached after a given number of iterations, then we will force the procedure to converge. Then value of N will be reduced by 1 and we repeat the above procedure. In the end, N is reduced to 0 which means there is only one active symbol in each lexicon. Then ambiguity is eliminated in all lexicons.

In sentence confabulation, the excitation level of a candidate is the weighted sum of excitation levels of active symbols in other lexicons. Intuitively, however, different source lexicons do not contribute equally to a target lexicon. For example, the lexicon right next to an unknown word obviously gives more information in determining the unknown word than the lexicon that is five words away. Thus the significance of a KL can be measured by weight and quantified by the

Algorithm 4. Baseline sentence confabulation recall algorithm

Input: *an ambiguous sentence* $\mathcal{S}$, *predefined maxAmbiguity, maxIteration*
Output: *a confabulated sentence* $\mathcal{S}'$
for each *known lexicon** $l_k \in \mathcal{S}$ **do**
  set symbol $s \in l_k$ *active*
end for
$N \leftarrow$ *maxAmbiguity*
while $N > 1$**do**
  converged $\leftarrow$ *false*
  iterationCount $\leftarrow 0$;
  while ¬*converged* **do**
    for each *unknown lexicon* $l_u \in \mathcal{S}$ **do**
      for each *symbol* $s \in l_u$ **do**
        calculate el(s)
      end for
      sort(l_u)
      for $i \leftarrow [0,N-1]$ **do**
        set symbol $s_i \in l_u$ *active*
      end for
    end for
    iterationCount $\leftarrow$ *iterationCount +1*
    if *active symbol set* $\mathcal{C}$ *unchanged*
      or *iterationCount* $\geq$ *maxIteration* **then**
      converged $\leftarrow$ *true*
    end if
  end while
  $N \leftarrow N - 1$
end for
update lexicons to $\mathcal{S}'$
output $\mathcal{S}'$

*lexicons who has only one symbol candidate are denoted as known lexicons, others are unknown lexicons.

mutual information(MI) [38] Mutual information of two random variables is a measure of variables' mutual independence, calculated as

$$I(A;B) = \sum_{b \in B} \sum_{a \in A} p(a,b) log \left(\frac{p(a,b)}{p(a)p(b)} \right)$$

where *A* is the source lexicon and *a* represents symbols in *A*; *B* is the target lexicon and *b* represents symbols in *B*. $p(a,b)$ is the joint probability of symbol *a* and *b*; $p(a)$ and $p(b)$ are the margin probability of symbol *a* and *b* respectively. $I(A;B)$ is nonnegative. The value of $I(A;B)$ will increase when the correlation of symbols in lexicon *A* and *B* get stronger. We defined the weight of KL (i.e. w_{kl}) from A to B as positive linear function of MI of A and B.

The sentence confabulation model in Algorithm 4 considers all initial symbols equally

possible. In reality, we know that some words are more likely than others from the given image. To incorporate the image information with sentence confabulation, we consider the BSB convergence speed during the confabulation process, and modify the excitation level calculation of a word symbol *t* as follows,

$$el(t) = \alpha P_{BSB}(t) + \beta \sum_{k \in F_l} \left[w_{kl} \sum_{s \in S_k} el(s) \ln\left(\frac{P(s|t)}{p_0}\right) + B \right]$$

In above equation, variable $P_{BSB}(t)$ is the excitation to *t* from the BSB layer, which is calculated as:

$$P_{BSB}(t) = \frac{1/(N_{BSB}(t) - N_{min})}{\sum_t 1/(N_{BSB}(t) - N_{min})}$$

where $N_{BSB}(t)$ is the BSB convergence speed of *t*, N_{min} is the minimum convergence number that is possible for BSB engines, α and β are coefficients that adjust the weight of BSB (i.e. image) information and confabulation (i.e. language) information, $\alpha + \beta = 1$. In general, we should increase the value of α and decrease the value of β when the image has high quality and vice versa.

As Fig. 24 shows, the idea of designing sentence confabulation layer is the same as that for word confabulation layer. The sentence scheduler will collect word candidates to reassemble a sentence from the MPI communicator when receiving any words. Once any completed sentence is collected, the sentence will enter the scheduling queue. The confab threads will de-queue the sentences to run Algorithm 4. The sentences will be confabulated to remove their ambiguity and sent to Result queue. MPI communicator will send the result to the next layer. Various different optimizations of this module are presented in [39].

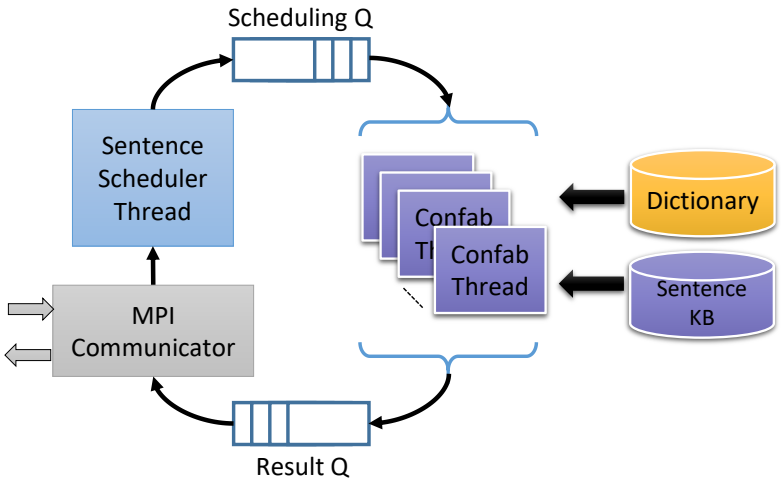


Fig. 24 . Sentence confabulation architecture

3.4.5 Result Gather Layer

Finally, the result gather module collects the out of order results from all sentence modules and saves them to files in sentence order.

This configurable ITRS pipeline provides the flexibility to maximize resource utilization and maintain a constant throughput in a heterogeneous cluster.

3.5 ITRS Performance Model

This configurable ITRS pipeline provides the flexibility to maximize resource utilization and maintain a constant throughput in a heterogeneous cluster. To build an analytical model, we collect the CPU level thread concurrency T and the memory bandwidth M needed to run every configuration c of each module along with its run time τ. To enable this data collection, each process along with its set of parameters must be analyzed independently i.e. $\forall c \in m$. The data is collected after the modules have been initialized as this kind of behavior is not recurrent while the application is processing inputs. For the case of ITRS, we select a set of input images which are a

representative of the type of inputs that will be processed then, we configure ITRS with one module in each layer to tap all MPI messages to a file. After this run, all communication between layers is recorded to files enabling each module to run independently as standalone modules. During a standalone run of any module m the saved messages from file are used as input to the module. In a standalone run all input is available beforehand for every module therefore, the module operates at maximum input capacity. The value of τ recorded is the minimum run time achievable for the given configuration c. The messages saved to the file consists of workloads to every module m while processing multiple number of pages. Therefore, the values of T, M and τ are normalized to number of pages to get per page concurrency, memory bandwidth and runtime.

We run each configuration c in 3 modes, once to collect the runtime of the configuration τ, for the next two modes the modules are launched with Intel VTune Amplifier analysis tool to record the CPU level thread concurrency T and the memory bandwidth M. Each of these configurations in every mode is executed 3 times to collect the average values. Therefore T, M and τ represent the average per page values.

3.6 Experiments and Results

We run the experiments on Intel Xeon and Nvidia GPU cluster and another cluster with Intel Xeon and Intel Xeon Phi (KNC and KNL architectures) machines. To show the effect of micro pipelining, each module is analyzed individually for different configurations and for different number of inputs. We can see the effects of pipelining in Fig. 25, as the input increases the run time decreases. Different configurations correspond to increasing number of threads in these experiments. BSB plot, Fig. 25 (b) is showing comparison based on hardware platform instead of configurations because it is optimized for fully utilizing the given amount of accelerator resources

allocated all the time. In Fig. 25 (a), cnfg_1 to cnfg_3 corresponds to a total of 6, 7, 13 threads across all thread pools respectively. For Fig. 25 (c) and (d), cnfg_1 to cnfg_4 correspond to 1, 2, 4, 8 confab threads respectively. From these plots, we can clearly see that BSB requires the most compute resources.

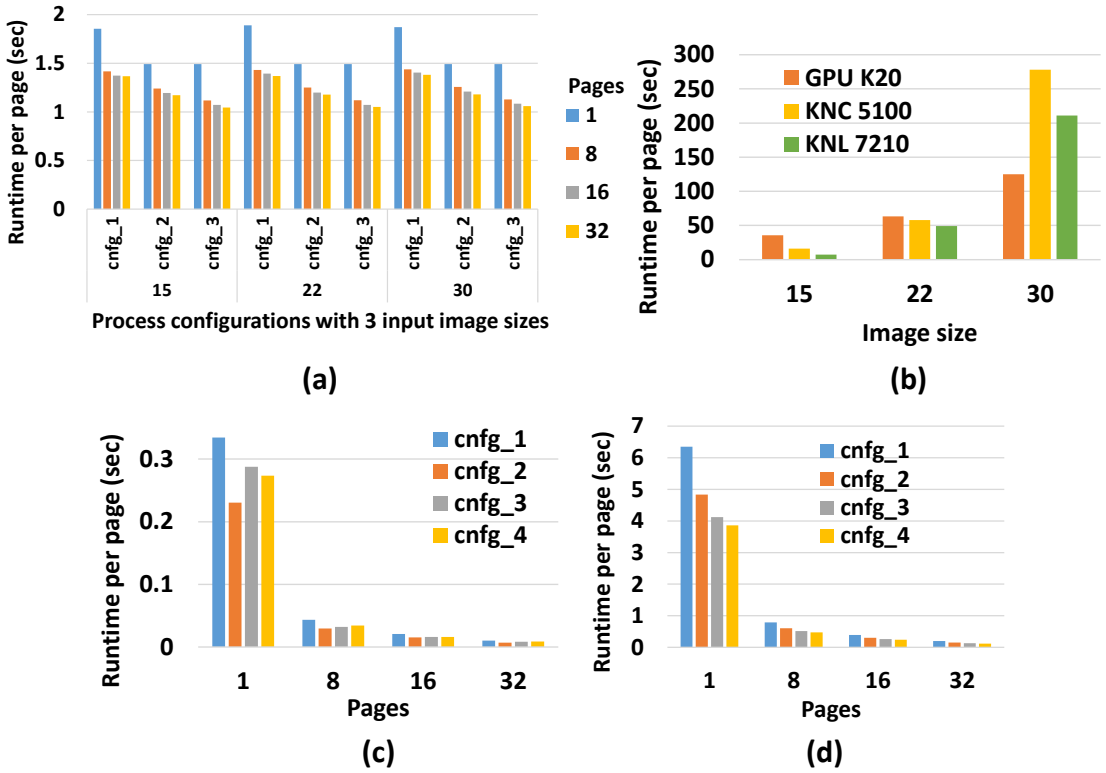
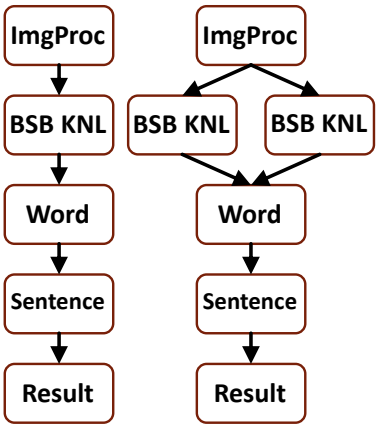


Fig. 25. Micro pipeline performance (a) Image processing, (b) BSB, (c) word confabulation, (d) Sentence confabulation

We run several experiments to demonstrate the effectiveness of scaling and macro pipelining. Fig. 26 shows STGs for a cluster with one Xeon node and one KNL node, all the modules are mapped to the Xeon node except BSB which is mapped to KNL node. We run these STG for a character resolution of 15x15 for pattern matching. For this resolution only partial resources on the accelerator is utilized, therefore a second instance of BSB can be mapped. The results for both the STGs is shown in Fig. 26 (c). We can see the macro pipelining effect for STG (a) however for STG (b) the resources are saturated, even though there is significant performance gain the

pipelining effect is not visible as BSB is still the bottleneck. To see significant performance gain due to pipelining enough BSB modules must be instantiated such that the bottleneck is eliminated.



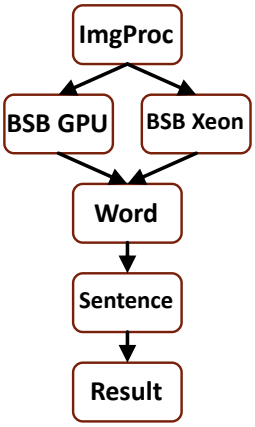
(c)

Num Pages	STG – (a) Runtime (sec/pg)	STG – (b) Runtime (sec/pg)	Performance gain
1	5.4490	2.8420	1.92
8	5.0365	2.9999	1.68
16	4.9609	3.0158	1.64
32	4.9841	3.0087	1.66

Fig. 26. (a) & (b) Accelerator sharing (c) Results demonstrating macro pipelining effect

To demonstrate the effect of distributed load balancing we obtain a STG using SLP as shown in Fig. 27 (a) for a cluster with 2 Xeon nodes, with one node having a GPU. This experiment is run with dynamic load balancing enabled and once with it being disabled. The table in Fig. 27 shows the runtime per page including pipeline fill time for both the cases. Due dynamic load balancing using a slower BSB module provides an improvement over not using the slower module at all. In this case BSB Xeon module is 7.48 times slower than BSB GPU module. We found that the experiments with flow control disabled resulted in poor reliability as the downstream modules were prone to buffer overflow errors as in this case the upstream forwards its messages irrespective of congestion in downstream modules. Due to structure based scheduling no extra effort was required to maintain accuracy of results with changing topology across various configurations. However, as the input resolution for BSB increased for 15x15 to 22x22 to 30x30 pixels per character the overall result accuracy increased but with an overhead of

extra compute time.



(a)

Load balancing	% load shared		Runtime (sec)
	BSB GPU	BSB Xeon	
Enabled	84.6	15.4	17.46
Disabled	50	50	53.11

(b)

Fig. 27. Dynamic load balancing (a) STG, (b) Results

To validate the proposed SLP framework we use a GPU cluster. Using SLP we generate several STGs and their corresponding throughput predictions. Then we run those STGs on the cluster and measure the throughput achieved on the cluster. These validation results are tabulated in TABLE II along with the ELPC results for comparison. We use the empirical performance data of modules for ELPC instead of normalized values as proposed by the authors for better comparison with SLP. We can clearly see that the predicted and achieved throughput are very well matching. The % error in prediction is also tabulated. The experiments 1 to 5 are running with character resolution of 15x15. Experiments 6 and 7 are running with character resolution of 22x22 and 30x30 resolution. Also, the number of nodes progressively increase in experiments 1 through 4. For these experiments BSB Xeon configurations are not included.

TABLE II. Validation results

Sl. No.	Nodes in cluster	SLP					ELPC				
		Modules assigned	Nodes used	Predicted throughput	Throughput on cluster	%error	Modules assigned	Nodes used	Predicted throughput	Throughput on cluster	%error
1	1	5	1	0.0281	0.0277	**1.5849**					
2	2	6	2	0.0562	0.0554	**1.4188**					
3	3	7	3	0.0843	0.0836	**0.8045**					
4	4	8	4	0.1124	0.1122	**0.1449**					
5	5	9	5	0.1405	0.1439	**2.3941**	5	5	0.0281	0.0276	1.6285
6	5	9	5	0.0792	0.0826	**4.2712**	5	5	0.0158	0.0157	0.7198
7	5	9	5	0.0400	0.0413	**3.4158**	5	5	0.0080	0.0080	0.0629

For experiments 1 to 4 ELPC cannot produce a solution as it doesn't model simultaneous resource sharing for stream processing. Fig. 28 shows the resource utilization and throughput gain as the application scales on the GPU cluster. Since ITRS has a bottleneck at the BSB layer, SLP rightly adds more modules to the bottleneck layer as more resources are available. We see that only BSB GPU modules are added to STG as additional resources are available this can be inferred by looking at curves SLP Avg. modules per node and SLP Avg. modules per layer. Since ELPC doesn't scale the, average modules per layer and average modules per node is always 1. Fig. 28 also shows the throughput gain achieved by SLP over ELPC model.

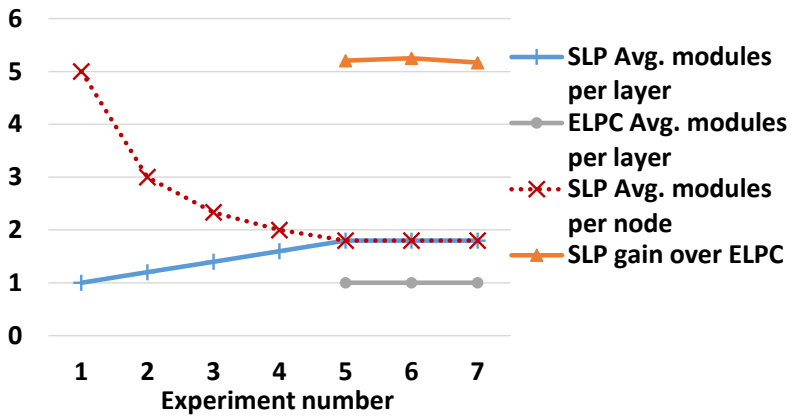


Fig. 28. Resource utilization efficiency and throughput gain of SLP over ELPC

Fig. 29 shows the scaling achieved through SLP as available resources increase along with the throughput gain of final evolution over the MFS solution.

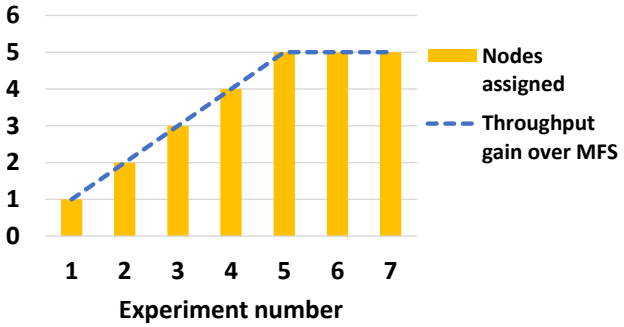


Fig. 29. Scaling with SLP

Fig. 30 compares the % error across different experiments. The average error of SLP prediction is 2% and ELPC prediction is 0.8%. The variance in error is due to the dynamic load balancing of the application and as the application scales, the messages between first 3 layers of ITRS increase due to quantized message sizes between them. This can be easily overcome by having dynamic message sizes like the rest of the layers, however at the same time this is not a significant issue.

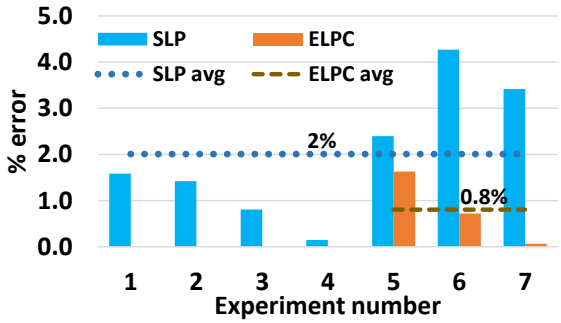


Fig. 30. Experimental vs analytical throughput prediction error

Till now we discussed ways of implementing complex applications, especially brain inspired algorithms on heterogeneous HPC framework in a large scale distributed manner. From this point forward we focus our attention towards achieving the same goals of implementing brain inspired inference models on very efficient and ultra-low power paradigm of biologically plausible spiking neural networks.

4 SPIKING NEURAL NETWORKS WITH DISTRIBUTED ONLINE LEARNING

HPC setup for implementing brain inspired algorithms have the benefit of utilizing todays readily available computing infrastructure. However, they are very inefficient compared to the biological brain. The brain is very efficient in terms of computation and power utilization. This is possible due to massively parallel computation being performed by the vast number of neurons while consuming very little energy due to the spiking nature of communication between them. However, the ability of neural networks to perform pattern recognition, classification and associative memory, is essential to applications such as image and speech recognition, natural language understanding, decision making etc. This is the inspiration behind spiking neural networks (SNNs). In SNNs, information is encoded as sparsely distributed train of spikes, which allows learning through the spike-timing dependent plasticity (STDP) property. SNNs can potentially achieve very large scale implementation and distributed learning due to the inherent asynchronous and sparse inter-neuron communications. SNNs are capable of representing much richer information as it incorporates relative spike timing along with the neuron state and the synaptic weights for computation.

There are many kinds of neural network architectures and learning algorithms proposed. For example Auto encoder, multilayer perceptron, deep learning, convolutional neural network etc., these are all computationally intensive when compared to SNN. The SNN has the potential to be very efficient as each neuron works asynchronously in an event-driven manner and with sparse spiking pattern. Moreover, fully distributed STDP learning [10] can only be achieved on SNNs, which updates synaptic weight based only on local information of individual neuron. The emerging stochastic SNN that generates spikes as a stochastic process is not only more

biologically plausible [11] but also enhances unsupervised learning and decision making [12] [13]. It further increases the fault tolerance and noise (delay) resilience of the SNN system since the results no longer depend on the information carried by individual spike but the statistics of a group of spikes.

Majority of the neuron models used in existing SNNs are not stochastic. Active dendrite and dynamic synapse with an integrate and fire neuron model is proposed for character recognition [4]. Spiking self-organizing maps using leaky integrate and fire neurons for phoneme classification is presented in [5]. They use this model to account for temporal information in the spike stream. Work presented in [7] uses Siegert approximation for integrate and fire neurons to map an offline trained deep belief network onto an event-driven spiking neural network suitable for hardware implementation. They demonstrate that the system is able to recognize digits in the presence of distractions and noise.

A large-scale model of a hierarchical SNN that integrates a low-level memory encoding mechanism with a higher-level decision process to perform visual classification task in real-time is implemented [8]. They model Izhikevich neurons with conductance-based synapses and use STDP for memory encoding. Stochastic nature in spike patterns has already been found in lateral geniculate nucleus (LGN) and primary visual cortex (V1) [11]. Ignoring the randomness in neuron model not only limits its effectiveness in sampling and probabilistic inference related applications [40] [41], but also reduces its resilience and robustness. This paper presents a STDP learning-enabled stochastic SNN for high noise tolerance.

In order to apply large scale SNN to machine learning applications, simple neuron models should be adopted, which are biologically inspired but not biologically realistic. The model should support efficient learning and recall while at the same time facilitate parallel and

distributed implementation. The neuron models are presented in this chapter which will be used in rest of the work to build complex networks.

4.1 BAYESIAN NEURON MODEL

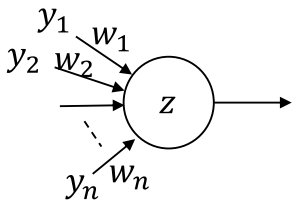


Fig. 31. Generic neuron model

We extend the generic Bayesian neuron model proposed in [42] for scalable and distributed computing purpose. This model supports recall and online learning using STDP. We use this Bayesian neuron model for building inference networks. This section discusses key background details of this model along with online STDP Learning. The details of a generic neuron model are shown in Fig. 31. In the neuron model, the membrane potential $u(t)$ of neuron Z is computed as.

$$u(t) = w_0 + \sum_{i=1}^{n} w_i \cdot y_i(t) \qquad (1)$$

where w_i is the weight of the synapse connecting Z to its i^{th} presynaptic neuron y_i, $y_i(t)$ is 1 if y_i issues a spike at time t, and w_0 models the intrinsic excitability of the neuron Z. The stochastic firing model for Z, in which the firing probability depends exponentially on the membrane potential, is expressed as

$$prob(Z\ fires\ at\ time\ t) \propto \exp(u(t)) \qquad (2)$$

In Eqn.(1), small variations of $u(t)$ resulting from the synaptic weight changes will have an

exponential impact on the firing probability, which is not desirable. To mitigate this effect a *range mapping function* is adopted. This function is a parameterized sigmoid function for representing more flexible S-shaped curves:

$$u'(t) = A + B/(1 + \exp(-(u(t) - C) \cdot D)) \quad (3)$$

The above equation has four parameters for shape tuning. Parameter: *A* provides Y-axis offset, *B* performs scaling along Y-axis, *C* provides X-axis offset and finally *D* performs scaling along X-axis. It maps a range of $u(t)$ to a different range $u'(t)$and the Out-of-range $u(t)$ to asymptotic values of the function. This makes sure that the membrane potential always lies within the dynamic range of the neuron. After mapping, $u(t)$ in Eqn.(1), should be replaced by $u'(t)$.

Two examples for Eqn. (3) are shown in Fig. 32. Curve (a) expands the input range (10, 20) to the output range (-10, 50). Any input value outside (10, 20) is asymptotically mapped to -10 or 50. Curve (b) compresses the input range (10, 60) to an output range (-10, 10) with asymptotic values for out-of-range inputs. For specific applications, the network topology is given and so is the possible range of the synaptic weights. Therefore, it is not difficult to have a rough estimation of the maximum and minimum values of the accumulated inputs of a particular neuron and a range mapping function can be chosen accordingly.

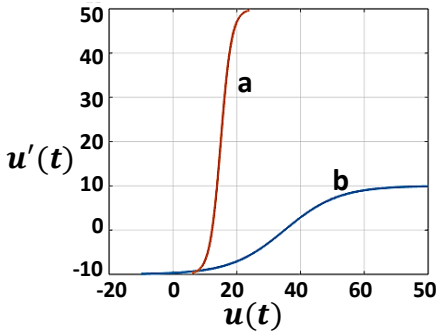


Fig. 32. Range modifier behavior

Learning involves updating the weight w_i of i^{th} synapse and the intrinsic excitation w_0 of the neuron. Their changes are calculated as below.

$$\Delta w_i = \begin{cases} ce^{-w_i} - 1, & if\ spike\ occured\ in\ STDP\ window \\ -1, & if\ there\ is\ no\ spike\ in\ STDP\ window \end{cases} \quad (4)$$

$$\Delta w_0 = e^{-w_0} \cdot z - 1 \quad (5)$$

The above delta changes are scaled with a constant learning rate to perform the final update. It can be proved [42] that based on this learning rule the w_i converges to the log of the probability that the presynaptic neuron y_i fired within the STDP window before neuron Z fires, and the firing probability of Z calculated by Eqn. (2) is the Bayesian probability of Z given the condition of its input neurons $y_1, y_2, \ldots\ y_n$ (i.e $P(Z|y_1, y_2, \ldots\ y_n)$.

To obtain Poisson spiking behavior, the method presented in [43] is adopted. The spike rate $\lambda(t)$ is an exponential function of the inputs, which is represented by Eqn.(4). To generate a Poisson process with time-varying rate $\lambda(t)$, the *Time-Rescaling Theorem* is used. According to this theorem, when spike arrival times v_k follow a Poisson process of instantaneous rate $\lambda(t)$, the time-scaled random variable $\Lambda_k = \int_0^{v_k} \lambda(v)dv$ follows a homogeneous Poisson process with unit rate. Then the inter-arrival time τ_k satisfies exponential distribution with unit rate.

$$\tau_k = \Lambda_k - \Lambda_{k-1} = \int_{v_{k-1}}^{v_k} \lambda(v)dv \quad (6)$$

To find the next spiking time v_k, a random variable is generated satisfying exponential distribution with unit rate, which represents τ_k. The integral in Eqn.(6) cumulates the instantaneous rates from Eqn. (2) over time until the integral value is greater than or equal to τ_k. Once this happens it implies that the inter-spike interval has passed and a spike is generated accordingly. In this way Poisson spiking behavior is generated based on the state of the neuron.

4.2 Spiking Rectified Linear Unit Neuron Model (ReLU)

From theory behind the Bayesian neuron model it is clear that the neuron is memory less and computation happens based on instantaneous rates. So, when this neuron is used in a network any weighted spike received by this neuron will have small effect on the overall firing rate. Hence the net effect of the weighted spike must be spread over time. This conversion mechanism is achieved by using a spiking Rectified Linear Unit (ReLU) neuron.

The ReLU function is defined as $Z = max(U_{th}, u(t))$ where Z is the number of output spikes, U_{th} is a constant threshold, and $u(t)$ is the membrane potential of this neuron calculated as $u(t) = u(t-1) + \sum_{i=1}^{n} w_i \cdot y_i(t) - U_{th}$. In other words, the membrane potential of a ReLU neuron accumulates every weighted input spike and discharges it over time resembling a burst firing pattern. In our implementation, the spiking threshold U_{th} is set to 1, and after each spike generation, the membrane potential is reduced by the threshold value. This makes sure that accumulated membrane potential is discharged faithfully over time.

4.3 Winner Takes All

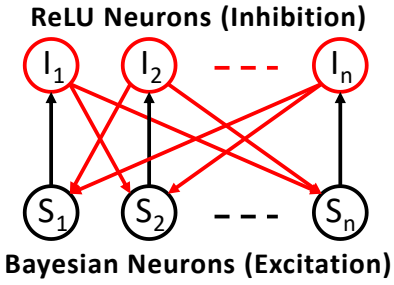


Fig. 33. Winner take all circuit

Fig. 33 shows a neural circuit to laterally inhibit a group of Bayesian neurons in a *winner take*

all (*WTA*) manner. WTA circuit is a recurrent network where a set of neurons compete with each other for activation. Each Bayesian neuron has an associated ReLU neuron, which collects and integrates input from other Bayesian neurons within this competing set and convert the accumulated signal into a sequence of inhibition spikes. Hard or soft WTA behavior can be achieved based on the degree of inhibition delivered. *Hard WTA* happens when the inhibition is strong such that it brings down the firing rate of the non-preferred Bayesian neurons to zero, resulting in only one neuron with highest excitation being active. On the other hand, if plural voting action is required within the set, the degree of inhibition is tuned to be moderate. This makes Bayesian neurons fire with different stable rates which is, *soft WTA* behavior where firing rate is proportional to their relative excitation levels.

5 HIGH PERFORMANCE SIMULATOR FOR SPIKING NEURAL NETWORKS

Several studies have been performed to confirm that most of the perceptual and motor tasks performed by the central nervous system are stochastic in nature, which can be modeled in a Bayesian framework [44] [45] [46] [42]. The decision-making process involves combining the priors with noisy information to compute predictions. SNNs built using these models have shown to perform inference based decision making. The emergent behavior of the model is not derived by mimicking the biological process in the neuron, but instead model the observed computational behavior. Hence these models are non-biologically realistic. There are several spiking neural network simulators available which support biologically realistic neuron models for large scale networks [40] [41] [47] [48]. However, there are no simulation tools available which can handle large scale SNNs with non-biologically realistic and mixed neuron models in an efficient manner. In this chapter, we propose a simulation tool architecture to enable development of SNNs with non-standard network topologies and neuron models. The network topology can be developed in traditional fashion with stacks of neuron layers or with any arbitrary topology including simulation of complex networks consisting of sub networks with arbitrary recurrent connections. We address the limitation while modeling these networks by enabling functionality to support stochastic behavior, making synapse modeling uniform so the neural computation is similar for inhibitory and excitatory case and enabling non-centralized control for neuron behavior.

Majority of available SNN simulators focus on biologically realistic neuron models, performing operational simulations and behavior characterizations. The NEURON simulation environment is primarily based on biologically realistic empirical models of neurons [40]. The GENESIS neural simulation system is another tool developed for computational neuroscience

[41]. It also implements biologically realistic neuron models for understanding biological networks and artificial neural networks. Brian is a Python based simulator [47], which supports rapid development of single-compartment neuron models and other complex models by defining the underlying behavior using differential equations. This to an extent supports non-standard neuron models for simulation by providing functionality to model non biologically realistic operations. NEST is another simulator for spiking neural network which focusses on dynamics and scale of the network rather than the exact morphology of the individual neuron [48]. This tool is efficient for scalable simulation of biologically realistic neuron models. In general these simulation tools compute differential equations in the neuron models which is computationally expensive. An event-driven simulator which exploits the sparse nature of neuron spikes to pre-compute look-up tables for characterizing synaptic and neuronal dynamics is implemented which improves simulation speed [49]. SpikeNET, usually runs very fast simulations but does not allow the simulation of very complex or biologically-realistic neural models, however this project is no longer supported [50].

These popular tools model the biological mechanisms and behaviors in detail, hence are bulky and complicated. It is not feasible to use them to simulate large-scale SNNs in a reasonable time frame. More computationally-efficient simulation tools and neuron models have been developed for large-scale SNN operation simulations. SpiNNaker, which is a low-power and parallel neuromorphic computing platform, can be used to simulate various types of neural networks with different kinds of neurons and connectivity patterns [51].

The bottleneck in wide spread adoption of SNNs is the lack of simulation tools to handle large-scale networks. Our proposed neuron model has features, such as uniform synapse connection and distributed learning, which facilitate large scale parallel implementation. In this

work, we develop Spiking Neural network Simulator (SpNSim), a flexible, multithreaded and vectorized spiking neural network simulator using C++. SpNSim has the ability to simultaneously simulate and train heterogeneous neural networks, i.e., networks consisting of different spiking neuron models with different behaviors including activation functions and STDP rules. This is a key feature in implementing complex neural networks with distinct subnetworks. The function of the simulator is validated using two networks representing two different applications from unsupervised feature extraction to inference based sentence construction.

5.1 ARCHITECTURE

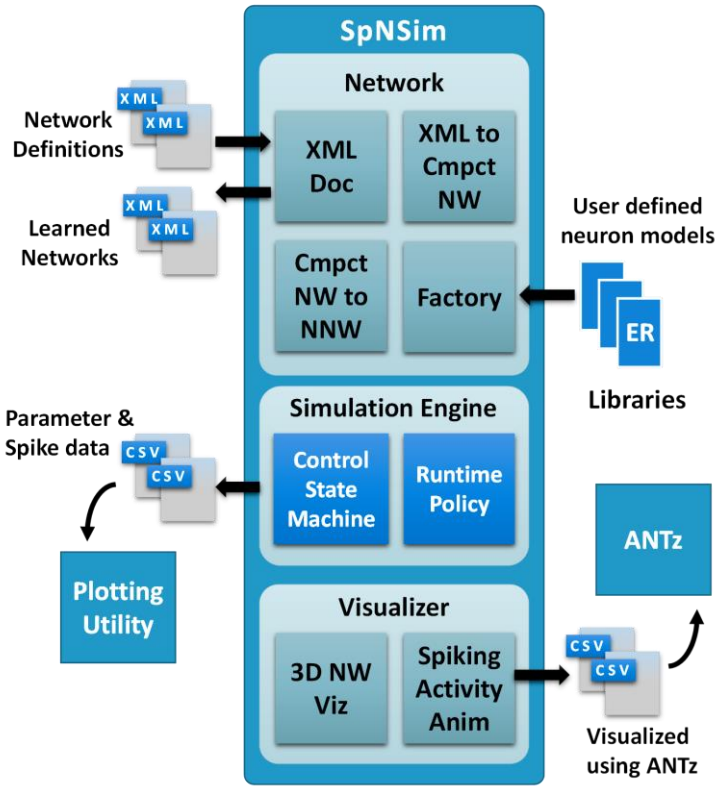


Fig. 34. SpNSim architecture

The SpNSim is designed to be modular and extendable. Its overall architecture is shown in

Fig. 34. There are three main layers in the design. First layer is Network layer where network definitions are read from user-provided XML files and neural networks are created. It also writes the trained networks back to XML files. Internally the network representation is maintained as a 3D graph in a Cartesian coordinate system, with vertices representing neurons and edges representing connections. The second layer is the simulation engine, which takes care of simulating the network in a multi-threaded environment. Finally, the visualizer layer helps in debugging and rendering the complex SNNs in 3D. All the subcomponents of SpNSim are described in detail in the following sections. The simulation treats time in discrete time steps called ticks.

5.2 Evaluation Routines for Neuron Model Simulation

We use interface class to make developing neuron models flexible so that any kind of behavior can be integrated. Neuron models are represented as evaluation routines (ER). ER provides platform for multi-threaded execution and thread synchronization. By default only one thread per ER is assigned, but based on the model requirement it is scalable. Each instance of ER is capable of holding data for any number of neurons of its type. The key advantage of this approach is that all data including weights and neuron parameters for numerous neurons of the same type are stored in arrays. These arrays are dynamically created and memory aligned to the processors vectorization boundary during initialization. The functions for the compute logic is developed such that there are no data dependencies across iterations of loops (i.e. no inter neuron data dependency), which is a prerequisite for enabling vectorization of code. The spike propagation is handled using pointers for quick communication. To avoid data dependency while computing current spike status we use two arrays, one is called current spike and the other is

called pending spike. For any given tick the status of the neuron is computed using current spike and the results are stored in pending spike to avoid data corruption as all the neurons are being evaluated in parallel. Later the pending spike will be copied to current spike. This constitutes to spike communication in a hazardless manner. Since all the compute logic and data is available in the ER, it is straightforward to optimize them for use with accelerators like GPGPU or Intel Xeon Phi. Though this kind of acceleration is possible, it not tested for current implementation.

5.3 RUNTIME POLICY

Runtime policy (RP) is a feature defined in the network definition file. This specifies dynamic behavior of the network, such as the starting and ending time of training and recall phases. Also at run time certain neuron parameters can be overridden for debug purposes or for modeling certain biological behaviors where the presence of neurotransmitters modulates the behavior of neurons for example providing reward and penalty behavior to neurons. The run time policies are defined as operations to be performed on the specified set of neurons. These operations are associated with triggers, which can be activated by certain conditions. Triggers can be of different types. Currently we have only implemented time triggers, with room for expansion to other kinds of triggers in the future. Time triggers are defined with activation expression, which always resolves to absolute simulation time. The expression can also be constructed based on other triggers plus relative time. Time triggers can also be defined as sequences or patterns, which resolve to a list of time steps. This rich way of defining triggers allows complex dynamic behavior of the network.

5.4 SIMULATION ENGINE

This module is core of the simulator. Once the network is created, a list of ER instances is

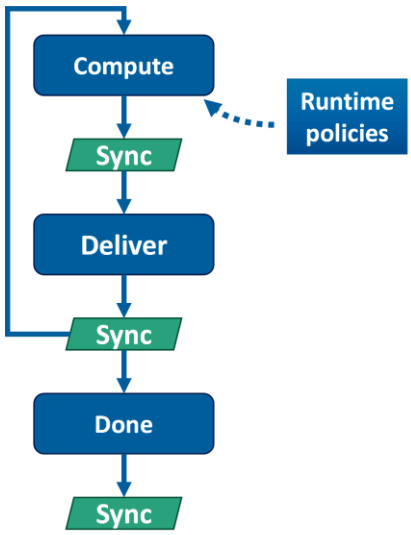


Fig. 35. Simulation engine control flow

registered with the simulation engine (SE). This process links up every thread from all the ERs with thread-safe blocking queues for two-way communication. Using blocking queue allows a thread to go to sleep while the queue is empty, thus freeing up resources for other operations. SE communicates with ER threads using command and response messages. Simulation engine implements a state machine with three states; Compute, Deliver and Done. Transition to next state is done only after a Sync operation where the responses of all the threads is received. The Sync operation enables the discretization of simulation time and also enables the computation of all neurons asynchronously with in each state.

The basic control flow is shown in Fig. 35. Simulation time is advanced in compute state and compute command is broadcast to all threads. After receiving their responses, the state transitions to deliver state. Here the output spike status generated during this tick is delivered as inputs to downstream neurons, which will be used for computation during next tick. This task primarily boils down to copying data from pending spike array to current spike array. Once the simulation time reaches the user-defined limit, it sends termination commands to all the threads. After receiving all responses, the simulation terminates. This ensures that all threads have safely

terminated and released all the resources back to the system.

The runtime policies including their triggers, operations and the set of associated neurons are resolved before the start of simulation. This information is registered with the SE. During the compute state SE checks for any triggered events. If any one of them are activated then the actions associated with the corresponding operations are included in the command messages to ERs, which encapsulates the list of specific neurons affected by that operation.

5.5 Network Specification and Creation

In biological nervous system, the neurons form well defined circuits performing specific tasks. To accommodate such complex neural circuits in SpNSim we define templates, which is a subnetwork comprising several neurons of any type with their associated connectivity. Instances of those templates can be placed at any given location in 3D space. Each neuron in that instance is referred hierarchically, using a concatenation of the instance name and its relative 3D location within the template. The intention of using template is to have a library of frequently used sub-networks and also to save trained networks, which can be reused as sub-circuits in more complex designs.

Two types of templates can be defined to realize neurons; group template and column template. Group template has a 2D structure, it defines placement of neurons in the X-Y plane. A column template has a 3D structure which is built by instantiating group templates along X, Y and Z directions. A column template also defines the connectivity among the instantiated group templates, hence creating a template of connected sub network. Since group instances in a column template results in building the column template, no physical neuron is realized until this column template is instantiated in the network. Like column templates the group templated can

be instantiated directly in the network to realize physical neurons.

The input of the simulator consists of one network definition file and any number of template definition files. All of them are specified in XML format. The template definitions can be included in the network definition file however, the use of template definition files provide the power of modularity and re-usability by lending support for developing a library of trained/re-usable sub-networks. The network definition file is responsible for instantiating all neurons within templates to build a network. The runtime policy is also specified in the network definition file for controlling the dynamic behavior of the network.

The connectivity among neurons can be specified as explicit connection or as a group. Explicit connectivity specifies connections from multiple source neurons to only one target neuron, whereas group specification makes multiple connections from specified list of source neurons to a list of target neurons. Depending on the requirement different patterns of connectivity can be assigned for example, full connectivity where all sources are connected to all targets or one-is-to-one connectivity where one source neuron only connects to corresponding target neuron in the list. Apart from this, probability values can be associated to the connectivity specification, so that links are established randomly. Weight patterns, including specific and random weight assignments, are defined for these connections as well. For example, a given weight is applied to all the connections or random weights with in the specified range are applied. Other connection parameters can be defined for example, an incoming connection to a neuron can have its learning mode enabled or disabled.

After running the simulation the learned network can be saved back to XML format. The network can be saved as network definition file or template definition file. The network definition file can be loaded back and run at a later time from the previous state. This allows snapshots of

simulation to be saved. If the network was saved as template definition file then it can be imported by any other network definition file and be used as trained sub-network.

SpNSim creates the network in a three-step process. In the first step we read all the definition files. These XML files are parsed and a XML tree structure is created. In the next step, using this tree a compact network representation is created. The reason behind creating this is to determine the total amount of memory required to build the SNN. The total memory requirement is not directly evident from the XML tree as templates can overlap in certain situations resulting in fewer neurons for such cases. Knowing the exact amount of memory is critical as this memory must be dynamically allocated such that it is memory aligned to the processors vectorization requirement. Finally, the actual spiking neural network is created. During the network creation process, first all the neurons are created then the connections are made, hence avoiding complex network graph traversals. To resolve name conflicts across network and template definition files, all file names in the project are required to be unique. In the internal representation of data elements all items are renamed with respect to their scope including file scope, hence making all names across files unique and a name resolution lookup is maintained to identify the right element.

Neuron types defined in the XML file result in ERs being created. These ERs behave as containers for neurons as described earlier. A neuron factory is used to create neurons in the appropriate ER based on the neuron type. If there are a large number of neurons of the same type, then extra instances of ERs can be created to increase the number of threads to evaluate them without modifying the underlying ER code.

5.6 3D VISUALIZER

Developing and debugging complex SNNs is a non-trivial task, especially when one must creatively imagine the network in 3D and provide the specification through XML. To facilitate this process, we use an open source 3D data visualizer called ANTz [52] to visualize the network and its spiking activities. SpNSim creates CSV files in the required format to represent the inferred SNN from the XML files. These CSV files are loaded on to ANTz for rendering the inferred network in 3D. All neurons are displayed as spheres and directed connections as cones with the base at the source neuron and the tip at the target neuron. Each neuron type is displayed with a different color for better understanding. SpNSim also records details of spiking activities in another CSV file, which can be used by the ANTz to animate spike generation and delivery.

5.7 PLOTTING UTILITY

SpNSim outputs spikes and neuron parameter data in CSV file format for the specified neurons. These can be analyzed using Microsoft Excel or using a rich plotting utility we have developed in MATLAB. This utility has the capability of showing raster plots for spikes and perform post processing like window analysis and signal to noise ratio analysis on these plots to determine the statistical behavior among the spikes. The neuron parameter variations can also be plotted for debugging and analysis purpose. The plotting utility is capable of displaying a set of weights over consecutive time steps resulting in an animated view of weight evolution during learning phase. This feature is particularly helpful in understanding the learning behavior of the network with respect to change in parameters.

5.8 Unsupervised Feature Learning and Extraction

The stochastic firing and STDP learning enables unsupervised feature learning and extraction, which is the function of the base layer in every convolutional network for image recognition. The MNIST dataset is used for this experiment to learn features of handwritten digits ranging from '0' to '9'. We use 2000 randomly selected samples from the training set to learn the features and tested against 2000 images randomly picked from the testing set. For all the experiments, we use binary MNIST images.

A convolutional neural network is constructed using the Bayesian neurons. Two different kernel sizes are used for the experiments, 5x5 and 7x7. For both kernels we set a stride of 2 pixels along X and Y directions. Each kernel is mapped to 9 features, implemented by 9 Bayesian neurons in the output layer. Each neuron of the Bayesian output layer is connected to all input neurons in its kernel. The input neurons perform population coding of input pixels, with two neurons representing black and white value of each input pixel. The neurons in the input layer fire, facilitating the Bayesian neurons to fire. Based on their relative spike-timing, the weight of the synapse is updated. A ReLU neuron based inhibition layer is attached to the output layer which realizes hard WTA function to ensure that only one feature will be activated for each kernel so that each Bayesian neuron learns a unique feature. The network setup of a 5x5 kernel size is shown in Fig. 36. The input layer consist of 50 neurons, the output and inhibition layers both have 9 neurons each. When an input neuron is active, it fires at 10% probability. The learning rate is fixed at 0.01, and the STDP period is 30 ticks for the experiments. The duration of STDP window is in the range of 10ms in a biological system.

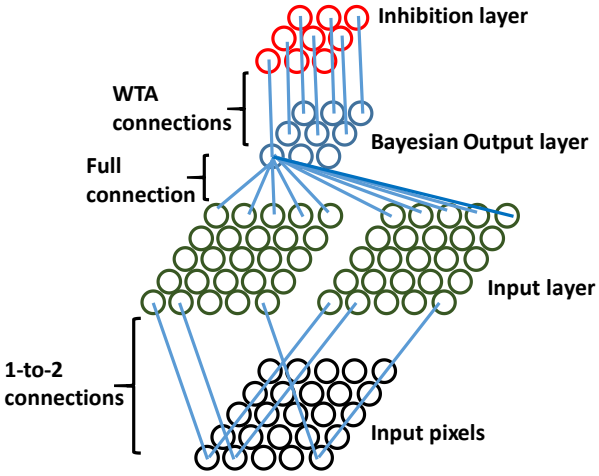


Fig. 36. Network structure (50x9x9)

The stochastic SNN performs learning and feature extraction similar to Convolutional Restricted Boltzmann Machine (CRBM) [53]. The same set of training and testing images is applied to an open source software implementation of CRBM. We found that they give comparable feature maps and filtered images as shown in Fig. 37. The support vector machine (SVM) classifier is used to check the effectiveness of the learnt features. Two different SVMs are trained and tested using features extracted from stochastic SNN and CRBM. The results show that the features extracted by stochastic SNN and CRBM can be used to classify with an accuracy of 94.4% and 94.45% respectively using 5x5 kernel, and 92.6% and 91.4% accuracy using 7x7 kernel, demonstrating the effectiveness of stochastic Bayesian neuron model with the results as shown in TABLE III. This accuracy is lower than the state-of-the-art results, which is 94% for SNN [7] and 99.18% for CRBM [53] as we are using the black-and-white images instead of greyscale images. We also observed that losing 50% of the connection will not cause notable performance degradation for large kernel. However, accuracy loss starts at 50% connections for small kernel. Finally, we expedite training by reducing the time that the training image is exposed to the system, and observe marginal impact.

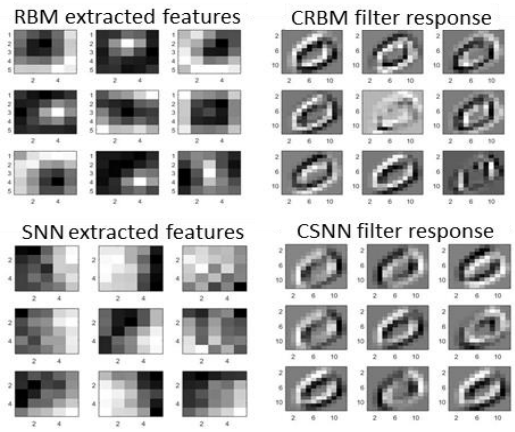


Fig. 37. Nine 5x5 extracted features and corresponding filter responses from our SNN and CRBM

TABLE III. Classification results

5x5 kernel (NW Size: 50x9x9)			7x7 kernel (NW Size: 98x9x9)		
Learning Time (ticks)			Learning Time (ticks)		
100	300	500	100	300	500
93.25	94.05	94.4	91.2	92.6	92.5
Connectivity %			Connectivity %		
50	70	100	50	70	100
91.7	92.35	93.25	91.2	90.25	91.2

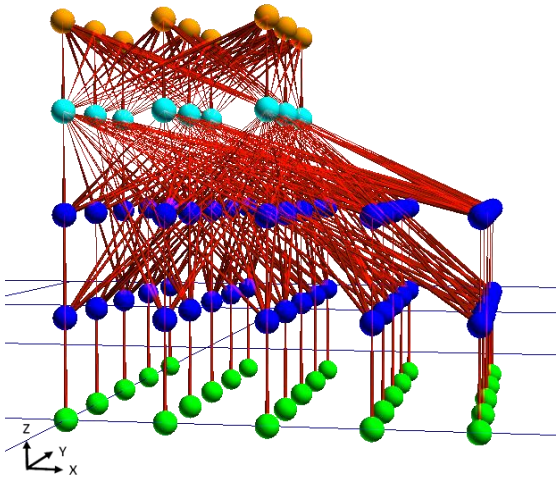


Fig. 38. 3D NW visualization of 9 feature learning of 5x5 kernel

The 3D visualization of the network learning 9 features from a 5x5 kernel is shown in Fig. 38.

The bottom layer of green neurons are input pixel neurons, they fire only if a bright pixel of the

image is exposed to the neuron. The next two layers of blue neurons correspond to input layers. One layer consists of neurons preferring black pixels and the other white pixels. The next layer with turquoise neurons are the Bayesian neurons and finally the orange neurons provide inhibition.

5.9 Confabulation Theory Based Inference

An inference network for sentence construction is created using Bayesian neurons. It consists of lexicons representing words and phrases. As shown in Fig. 33, a lexicon is a WTA subnetwork with Bayesian neurons for excitation and ReLU neurons for inhibition. Each Bayesian neuron represents a symbol, which in this case is a potential word or phrase at certain location of sentence. The synapses between neurons across lexicons are created based on the log conditional probability of the two connected words (phrases). All neurons are initialized with the same intrinsic potential (i.e. the same initial firing rate). The most strongly connected neurons resonate and enhance each other and at the same time inhibit the other neurons in the same lexicon. The network settles on contextually correct behavior and neurons with the highest firing rate in each lexicon marking the sentence that is grammatically correct and semantically meaningful. In this

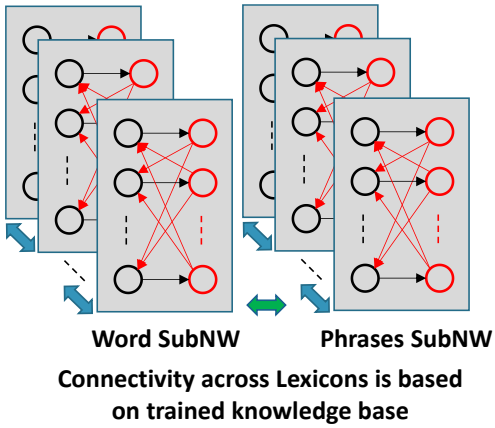


Fig. 39. Sentence confabulation network

application, the inhibition layer performs soft WTA. It has an advantage over hard WTA because symbols with lower excitation are not discarded, thus more information is retained during inference. Fig. 39 shows the network topology.

We randomly picked 45 sentences from document images. Fuzzy character recognition is performed on these images which results in multiple possible words for each word position as described in [2]. For example, given input candidates [{we, wo, fe, fo, ne, no, ns, us} {must, musk, oust, onst, ahab, bust, chat} {now, noa, non, new, how, hew, hen, heu} {find, rind, tina} {the, fac, fro, kho} {other, ether}], the SNN settles at a grammatically correct sentence as [we

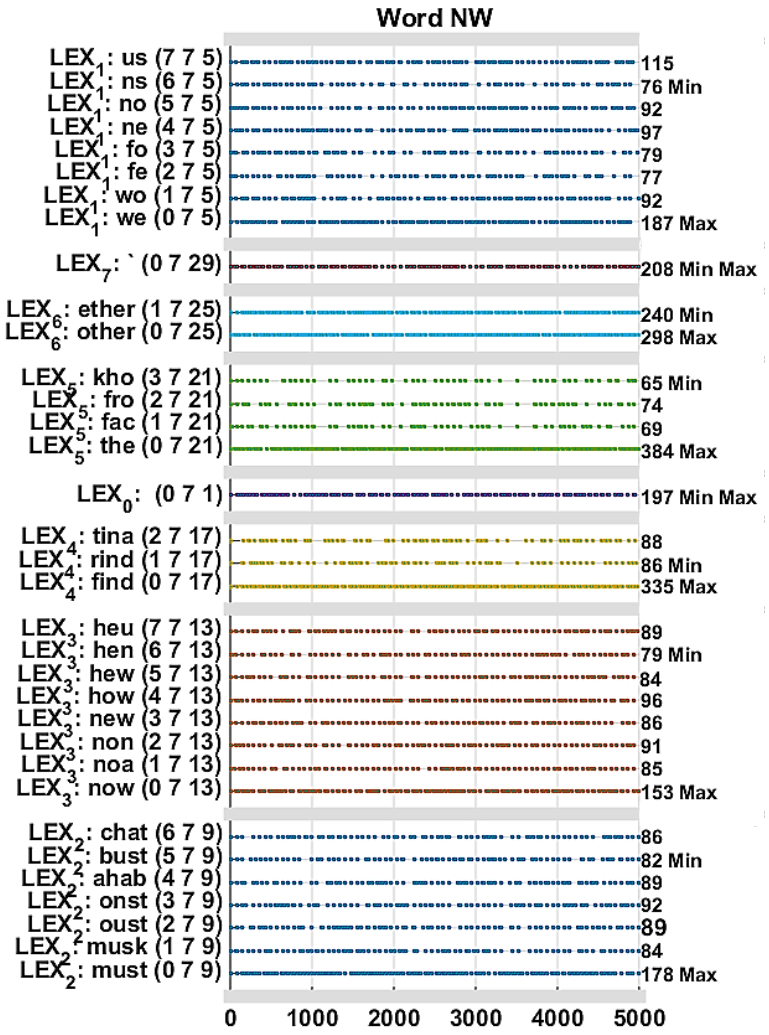


Fig. 40. Confabulation results raster plot

must now find the other]. The raster plot for this example is shown in Fig. 40. The labels along Y-axis are grouped by lexicon in the following format; the lexicon number, symbol represented by the neuron and the 3D (x y z) co-ordinate of the neuron in the network. The spike count for winning symbols is highest in the Lexicon, which is shown along the secondary Y-axis. The X-axis represents the simulation ticks. Lexicon-0 and lexicon-7 in the figure mark the beginning and end of the sentence. The average SNR across all lexicons is 2.57. Overall, the stochastic SNNs are able to construct correct sentences for 83.8% of the test cases.

6 LOW POWER NEURON MODEL FOR DIGITAL HARDWARE

SNNs can achieve ultra-low power consumption due to their sparsity and distributed nature. Several inroads have been made in SNN implementations; however, there is still a lack of computational models that lead to hardware implementation of large scale SNN with STDP capabilities. In this work, we present a set of neuron models and neural circuit motifs that form SNNs capable of in-hardware fully-distributed STDP learning and spiking based probabilistic inference. In this chapter, a highly scalable and flexible digital hardware implementation of the neuron model is presented. Functions such as Bayesian inference and unsupervised Hebbian learning are implemented on the proposed hardware SNN system to demonstrate the design's effectiveness in learning and inference.

The potential benefits of the SNN cannot be fully realized without dedicated hardware because full software implementations have high coordination overheads and are limited by the allowable degree of parallelism. Many traditional computational models of SNN are not designed to facilitate hardware implementation [2] [54]. They either consist of excessive physiological details [40] [41] or rely on centralized control to coordinate neurons [42]. Some recent research works on SNNs have been carried out from the hardware design perspective [7] [55] [51]. Novel hardware systems such as IBM's TrueNorth neurosynaptic processor has enabled breakthrough in design and applications of SNN. However, STDP learning has not been an integral part of the neuron model in these hardware systems. As a result, they do not support real-time in-hardware learning, which is critical when being applied to a dynamic environment or to satisfy the requirement to run multiple applications. Although hardware implementation of STDP learning has been discussed in several previous publications [56] [57], these works focus more on circuit

and device level analysis on how variable synaptic plasticity is achieved. They either did not demonstrate the ability of learning [57] or were applied only to small scale problems with linearly separable classes [56]. Furthermore, these implementations are either in analog domain or rely on certain non-linear properties of the device while no specific computational model was provided.

A dedicated hardware implementation of the SNN is a very attractive option for a large variety of applications due to its significant potential in energy efficiency. The biological neuron models are bulky and complicated, thus not suitable for large-scale implementations. Neurogrid, developed at Stanford University for simulation of biological brains [58], uses analog circuits to emulate the ion channel activity and uses digital logic for spike communication. BrainScaleS is another hardware implementation that utilizes analog neuron models to emulate biological behavior [59]. These implementations have been focusing on biologically realistic neuron models and are not optimized for large-scale computation. IBM has come up with the TrueNorth architecture which is digital and optimized for large-scale applications and contains 4096 cores with 256 neurons in each core [60]. None of the above hardware systems support real-time in hardware STDP learning. Several existing efforts address hardware implementation of the STDP function [56] [57]. Their main focus is how to use nonlinear property of resistive RAM or analog circuit to realize variable synaptic weights. [56] applies a ReRAM array to memorize the EEG signal of three vowels and [57] does not provide experimental data to demonstrate the circuit's ability to learn. Neither of them give details of their computational model.

In this chapter, we focus on a large scale digital hardware implementation of the stochastic SNN with biologically plausible in-hardware learning. An improved computational model of the stochastic SNN is presented. The model describes neuron behavior, STDP learning rules and network topology. A reference digital implementation of the neuron model is also provided,

which is highly scalable and flexible.

6.1 Recap of Bayesian Neuron Model

As described in chapter 4, in our model the excitatory and inhibitory inputs are treated in a uniform way since both are spiking based. Synapses with a positive weight induce excitation and synapses with negative weight provide inhibition. A range mapping function as described in Eqn (3) is used to maintain the neurons dynamic range. To sum up, the proposed neuron model handles all synaptic inputs uniformly, and its membrane potential can be constrained within a fixed range. Such regularity reduces the hardware implementation complexity. The range-mapping function also provides the means to adjust the membrane potential and consequently the firing rate across neurons without retraining the entire network. To obtain Poisson spiking behavior we adopt the method introduced in [43] as described in chapter 4.1.

6.2 Efficient Winner-Take-All Circuit

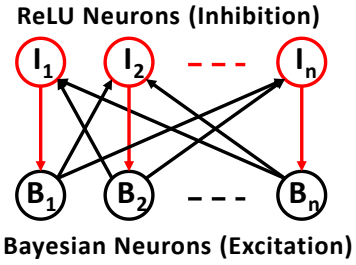


Fig. 41. Efficient winner-take-all circuit

Fig. 41 shows an enhanced WTA neural circuit to laterally inhibit a group of Bayesian neurons. In this circuit the ReLU neurons inhibit only their corresponding Bayesian neuron. This contrasts with the earlier proposed design shown in Fig. 33 where, the ReLU neurons inhibit all the lateral Bayesian neurons except their corresponding Bayesian neuron. The Bayesian neuron is

memoryless by design and computes instantaneous probabilities. Therefore, in the former case all the inhibition which is delivered in the same instant will have a small impact as there is limited cumulative effect over time. This drawback is overcome in the latter circuit where all the accumulated inhibition is delivered faithfully over time hence more accurate inference is possible. The model avoids centralized generation of the inhibition signal, therefore there is no need to synchronize neuron activities. This makes hardware implementation simple and distributed. Henceforth, we refer the enhanced WTA circuit as WTA circuit for simplicity.

As described earlier, Hard or soft WTA behavior can be achieved based on the degree of inhibition delivered. *Hard WTA* happens when the inhibition is strong such that it brings down the firing rate of the non-preferred Bayesian neurons to zero, resulting in only one neuron with the highest excitation being active. Hard WTA can be used for unsupervised feature extraction for enabling each neuron to learn a unique feature. On the other hand, if plural voting action is required within the set, the degree of inhibition is tuned to be moderate. This makes Bayesian neurons fire with different stable rates which is the *soft WTA* behavior where firing rates are proportional to their relative excitation levels. Soft WTA helps to retain more information in probabilistic inference. The WTA circuit is the basic building block for our SNN.

6.3 Hardware Architecture of Digital Neuron Model

Keeping reliability, scalability and flexibility as the primary focus we choose digital implementation over analog. This section presents a reference design of a hardware neuron. Two functions are supported by the hardware, (1) membrane potential update and spike generation, also referred as the “recall” function, (2) synapse weight update based on STDP rule, also referred

as the "learning" function. For lower hardware cost and power consumption, each set of hardware is used to evaluate the recall and learning function of multiple neurons in the SNN in a pipelined manner. We refer to the hardware implementation as *physical neurons*, and the neurons in the computational model as *logical neurons*. A core based architecture is employed where each core has a physical neuron which is time multiplexed to handle 256 logical neurons. Each core is also associated with a network on chip router which interfaces with other cores. Each core also has a crossbar whose inputs are axons receiving spikes from the router. The output of the crossbar are the dendrites feeding the logical neurons. This core based architecture is similar to the one employed by the TrueNorth chip [60]. In this chapter, only the physical neuron design which supports online learning is discussed.

Approximation and resource sharing techniques are adopted in the proposed design to reduce hardware complexity. Instead of directly implementing Eqn.(3), we approximate it using a piecewise linear function. The asymptotic regions of the curve are approximated with constant values, while the rest of the curve is approximated with a straight line represented as $u'(t) = m \cdot u(t) + c$ where m is the slope of the line and c is the Y-axis intercept. In this way, the range mapping function can be implemented using a multiplier and an adder. The exponential relation between the firing rate and membrane potential (i.e. Eqn. (2)) is realized with an exponential lookup table, which reduces the compute and area requirement.

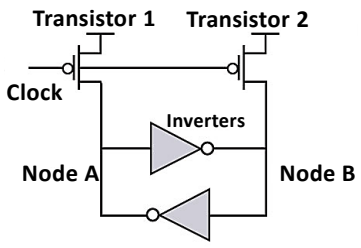


Fig. 42. Bistable 6-T random number generator design.

To generate the Poisson firing pattern as described by Eqn. (6), we use a geometric distribution instead of an exponential distribution as discrete values are required. A random number is drawn whenever a spike is generated. The traditional shift register based pseudo random number generator will require a significant amount of area and power with a limited degree of randomness. Motivated by the Intel's design for Ivy Bridge [61], we adopt a 6-transistor (6-T) random number generator as shown in Fig. 42, which comprises a bistable structure and two pull-up transistors. When the clock signal is in the low phase, both Node A and Node B are pulled up (close) to V_{dd}. When the clock signal is in the high-phase, both pull-up transistors are cut-off and the bistable begins evaluation phase. During evaluation, any noise or disturbance will drive the bistable out of the unstable equilibrium point (Node A = Node B = V_{dd}/2) and make one of them logic 1. Nodes A and B will become logic 1 with equal probability because the noise will incur equal probability of positive and negative effects on the node voltage. Extensive experiments have been conducted on this design, demonstrating its effectiveness and 5% tolerance level on process variations.

This random number generator is used to compute the geometric distribution which represents the next spike generation time. The significant values in a geometric distribution lie within a small range, hence the random number is directly used with a programmable mask for fine tuning. The exponential lookup table along with multiplier and an additional adder is used to realize Eqn. (4) and (5).

Two memory banks are needed in the hardware implementation. The first one is a configuration memory, which stores the parameters such as learning rate and the coefficients of the range mapping function. The second is the neuron status memory. It stores the weight and the accumulated inter-spike time (as calculated by Eqn. (6)) for each synapse and STDP window

status. Each neuron requires 806 bytes of neuron status memory and 13 bytes of configuration memory.

6.4 Dataflow Graph and Data Path Architecture

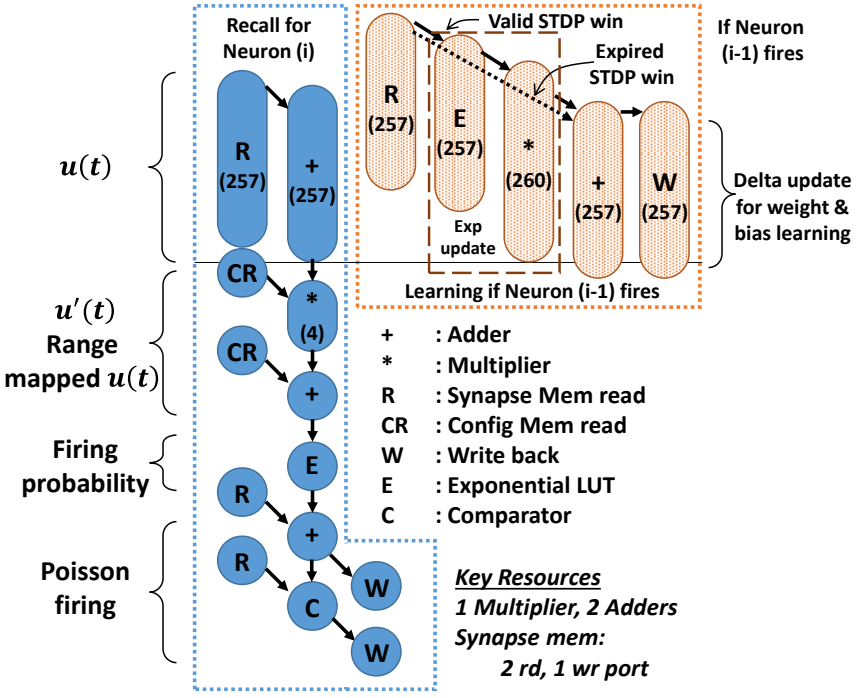


Fig. 43. Dataflow graph for pipelined recall and learning

Fig. 43 shows the data flow graph (DFG) of the learning and recall functions of a Bayesian neuron. The operations in blue are required for both learning and recall functions. They calculate the membrane potential and evaluate spike firing conditions. The weight of each input synapse is read from status memory and accumulated according to Eqn. (1). We assume that each neuron has a maximum of 256 input synapses. For simplicity, the DFG groups 256 memory read operations into one 256-cycle memory read, and 256 addition operations into one 256-cycle addition. Due to data dependency, the addition starts one cycle later than the memory read. The calculation of membrane potential $u(t)$ will then go through range mapping, spiking rate

generation and Poisson firing steps as indicated in the DFG.

The operations in orange are required only for the learning function. They will be executed only when learning is enabled and either a spike is issued within the STDP window or the STDP window has expired. In both cases, the original synaptic weight will be read out and the updated synaptic weight will be written back. If a spike is issued within the STDP window, the exponential LUT lookup and multiplication will be executed to calculate the Δw_i according to the first part of Eqn. (4). Otherwise, if the STDP window expires, the *Exp update* block will be skipped and a constant negative Δw_i will be used to update the synaptic weight according to the second part of Eqn. (4). No action will be taken in all other cases. Our simulation shows that these learning related operations are only executed with less than 17% probability during the learning stage. The fact that online learning is performed much less frequently compared to recall leads to lower total power consumption. Again, the operations performed on 256 synaptic weights (w_i) plus one intrinsic weight (w_o) is chained into a 257-cycle operation. To achieve high throughput, a 4-stage pipelined multiplier is used. As a result, it takes 260 cycles to process 257 multiplications in the data flow graph. Because the synaptic weight can only be updated after the condition for firing is evaluated, the learning function of the $(i\text{-}1)^{th}$ logical neuron can overlap with the recall function of the i^{th} logical neuron.

An analysis on the data flow graph shows that two adders and one multiplier are needed as computational resources. It also shows that the neuron status memory must have two read ports and one write port. With these resources the overall latency to evaluate the recall and learning functions of a logical neuron is 526 cycles and the throughput is 267 cycles per logical neuron. We define the time to evaluate all 256 neurons in a core as the *neuron evaluation cycle* (*NEC*). One NEC consists of 267*256 + 259 = 68,611 clock cycles.

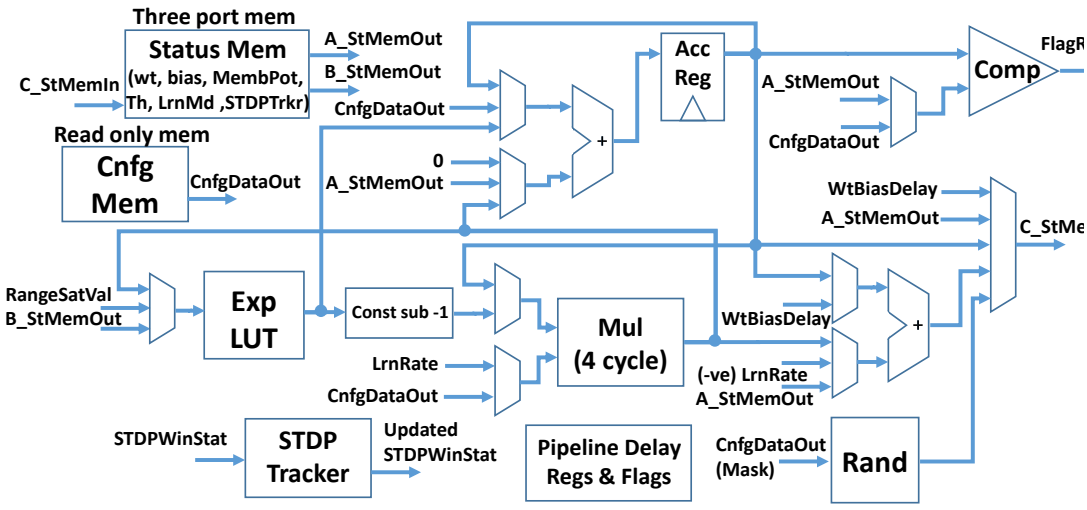
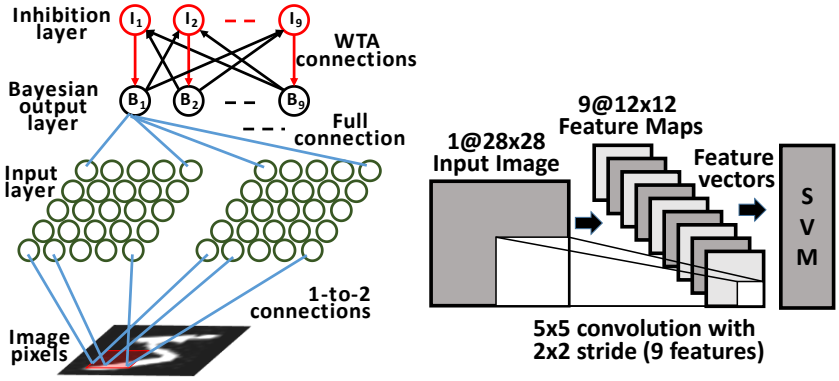


Fig. 44. Neuron datapath

Based on the above analysis, we developed a digital architecture with 16-bit fixed-point precision for the neuron model encompassing both recall and learning circuits. Fig. 44 shows the datapath of our design. The controller is divided into two state machines, one for recall and the other for learning. By disabling the learning module and stochastic firing function, the same design can be used to implement integrate and fire as well as the ReLU neurons.

To validate its functionality, we applied the neuron model on two different applications. In the first experiment, the neuron model is used to perform unsupervised feature learning and extraction of hand written digits. With this experiment, we will demonstrate the in-hardware learning capability of the neuron model. The potential tradeoff between hardware complexity using fixed point arithmetic and detection quality will be discussed. The second experiment demonstrates the model's capability of performing Bayesian inference, where it is applied for sentence construction with a learned natural language model. Function accurate C++ neuron model was developed and simulations were performed on networks using SpNSim to cross validate the hardware.

6.5 Unsupervised Feature Learning and Extraction



(a) (b)

Fig. 45. Network structure for (a) training and (b) testing

The stochastic firing and STDP learning enables unsupervised feature learning and extraction, which is the function of the base layer in every convolutional network for image recognition. The MNIST dataset is used for the proposed model to learn features of handwritten digits ranging from '0' to '9'. We use 2000 randomly selected samples from the training set to learn the features and tested against 2000 images randomly picked from the testing set. For all the experiments, we use binary MNIST images.

A convolutional neural network was constructed using the Bayesian neurons. Kernels with two different sizes are tested, 5x5 and 7x7. For both kernel sizes we set the X and Y strides to be 2 pixels. Each kernel is mapped to 9 features, implemented by 9 Bayesian neurons in the output layer. Each neuron of the Bayesian output layer is connected to all input neurons of the kernel. The input neurons perform population coding of input pixels, with two neurons representing black and white value of each input pixel. The neurons in the input layer fire, facilitating the Bayesian neurons to fire. Based on their relative spike-timings, the weight of the synapse is updated. A ReLU neuron based inhibition layer is attached to the output layer and implements the

hard WTA function to ensure that only one feature will be activated for each kernel and each Bayesian neuron learns a unique feature. The setup for learning and testing a 5x5 kernel is shown in Fig. 45. The input layer consist of 50 neurons, and the output and inhibition layers both have 9 neurons. When an input neuron is active, it fires at a 10% probability. The learning rate is fixed at 0.01, and the STDP period is 30 neuron evaluation cycles for the experiments.

TABLE IV. Classification results

5x5 kernel (Network Size: 50x9x9)			**7x7 kernel (Network Size: 98x9x9)**		
Learning Time (NEC)			**Learning Time (NEC)**		
100	300	500	100	300	500
93.25	94.05	94.4	91.2	92.6	92.5
Fixed Point Precision (bits)			**Fixed Point Precision (bits)**		
8(4,4)	16 (8,8)	32 (16,16)	8(4,4)	16 (8,8)	32 (16,16)
90.3	91.35	93.25	89.85	87.45	91.2
Connectivity %			**Connectivity %**		
50	70	100	50	70	100
91.7	92.35	93.25	91.2	90.25	91.2

The stochastic SNN performs learning and feature extraction functions similar to a Convolutional Restricted Boltzmann Machine (CRBM) [53]. The same set of training and testing images is applied to an open source software implementation of CRBM. We found that they produce comparable feature maps and filtered images as shown in Fig. 45. A support vector machine (SVM) classifier is used to check the effectiveness of the learnt features. Two different SVMs are trained and tested using features extracted from our stochastic SNN and CRBM. The results show that the features extracted by stochastic SNN and CRBM can be used to classify with 94.4% and 94.45% accuracy respectively using the 5x5 kernel, and 92.6% and 91.4% accuracy respectively with the 7x7 kernel. Please note that, although the state-of-the-art technique can recognize MNIST data with 99.2% accuracy [53], this is achieved using a multi-layer deep belief network with 60,000 training images. While ours has only one layer and trained using 2000 images. It is our next step to develop a multi-layer network using the stochastic SNN. The

accuracy of our SNN is close to the best results in [62], which is 95%. However, they use 6,400 excitatory neurons, which is 5 times more than ours. Furthermore, it memorizes the whole image, therefore it is hard to improve its accuracy by adding further layers; while ours is a convolutional network, which can be extended to a deep neural network.

Functional simulation of the hardware design was carried out to explore the tradeoff between cost and performance. TABLE IV compares the accuracy of pattern classification when different fixed point data precisions and different connection ratio between the input and Bayesian layers are used. As we can see, the quality of learned features (i.e. the classification accuracy) drops marginally when the data precision is reduced from 32 bit to 8 bit. We also observed that losing 50% of the connection will not cause notable performance degradation for the 7x7 kernel. However, accuracy loss starts at 50% connections for the 5x5 kernel. Finally, we expedite training by reducing the time that the training image is exposed to the system, with only marginal impacts.

6.6 Inference Based Sentence Construction

An inference network for sentence construction is created using the Bayesian neurons and stochastic SNN. It consists of lexicons representing words and phrases. As shown in Fig. 41, a lexicon is a subnetwork of Bayesian neurons for excitatory and ReLU neurons for inhibitory functions. Each Bayesian neuron represents a symbol, which in this case is a potential word or phrase at a certain location of sentence. The synapses between neurons across lexicons are created based on the log conditional probability of the two connected words (phrases). All neurons are initialized with the same intrinsic potential (i.e. the same initial firing rate). The most strongly connected neurons resonate and enhance each other and at the same time inhibit the

other neurons in the same lexicons they belong to. The entire network settles on contextually correct associations and neurons with the highest firing rate in each lexicon marking the sentence that is grammatically correct and semantically meaningful. In this application, the inhibition layer performs the soft WTA function. It has an advantage over the hard WTA because symbols with lower excitation are not discarded, thus more information is retained during the inference. Fig. 39 shows the network topology except that it uses the efficient WTA sub-networks

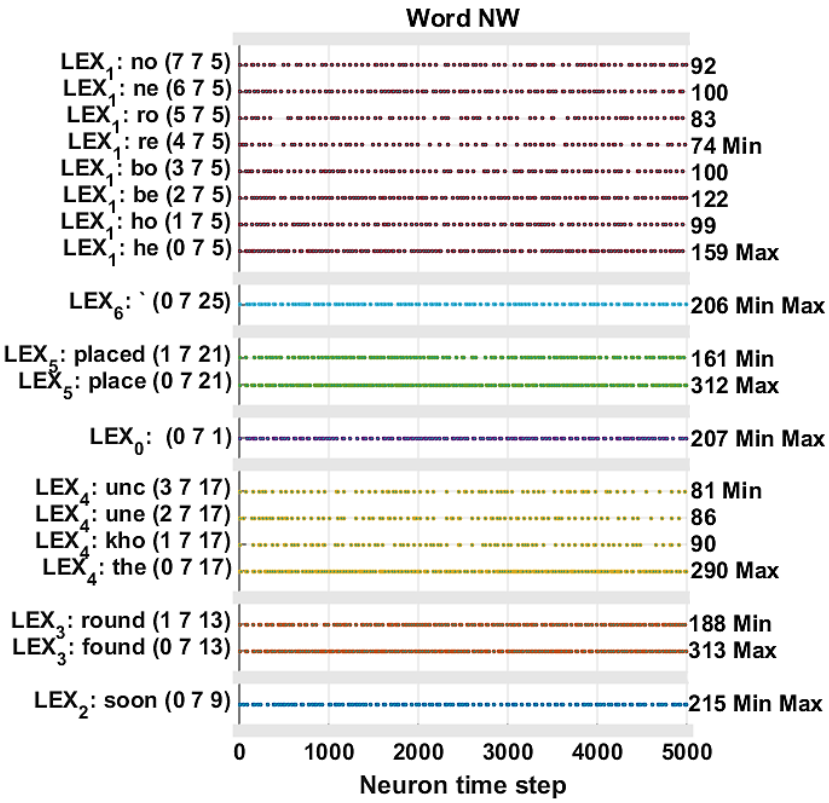


Fig. 46. Confabulation results raster plot

We randomly picked 45 sentences from document images. Fuzzy character recognition is performed on these images which results in multiple possible words for each word position as described in [2]. For example, given input candidates [{he, ho, be, bo, re, ro, ne, no} {soon} {found, round} {the, kho, une, unc} {place, placed}], the SNN settles at a grammatically correct sentence as [he soon found the place]. Fig. 46 shows the raster plot for one of the sentences. The

spike count for winning symbols is identified as “Max” in a lexicon. The stochastic SNNs are able to construct correct sentences for 83.8% of the test cases.

6.7 HARDWARE IMPLEMENTATION ANALYSIS

An RTL design of the neuron model was developed and verified through functional simulation. The design is synthesized using 45nm, 1.1V CMOS technology. Our primary focus is to minimize power and area. The RTL design is synthesized with these constraints and we use the CACTI tool to estimate the access time and power consumption of the memory blocks with high threshold devices (for low leakage power.) In order to achieve similar speed as a biological neural system, our target is to complete one NEC in 0.5ms. This converts to a maximum clock period of 7.3ns. Synthesis results show that the minimum clock period is 3.15ns, hence we are well within the target margin with an area of 4460 μm^2 for digital logic and 209KB memory for 16-bit data path. Details of these result can be found in [63].

Not all resources are used every clock cycle. In our experiments, typically the learning circuits are used 17% of time with sparse spiking pattern. Out of this, the multiplier is used for 3% of the time and the rest is spent on the adder for weight update. For recall operation, the multiplier is utilized for 1.5% of the time. Overall the multiplier is utilized for only 2% of the time, therefore consuming little dynamic power.

7 PROBABILISTIC GRAPHICAL MODEL MAPPING AS A SPIKING NEURAL NETWORK

Bayesian inference and belief networks are powerful tools for many applications, such as error correction, speech recognition, and image recognition. Recently deep belief networks have demonstrated amazing results in unsupervised feature extraction [64] and image recognition [53]. A stochastic SNN naturally implements Bayesian learning and belief propagation. In [42], the authors present a Bayesian neuron model and the STDP learning rule. It can be proven that based on given STDP learning rules the synaptic weight of a neuron converges to the log of the probability that the presynaptic neuron fired within the STDP window before post synaptic neuron fires, and the firing probability of the post synaptic neuron is its Bayesian probability given the condition of its input neurons.

Despite the simplicity of the SNN, it is not efficient when implemented on traditional processors with the Von Neumann architecture, due to the performance gap between memory and processor. The IBM Neurosynaptic System provides a highly flexible, scalable and low-power digital platform [60] that supports large scale SNN implementation. IBM's neurosynaptic processor called TrueNorth has 4096 cores and each core features 256 neurons and axons. The synaptic connections and their weights between axons and neurons are captured by a crossbar matrix at an abstract level. This abstraction is in the form of the programming paradigm for TrueNorth called Corelet [65]. Corelets represent a network on the TrueNorth cores by encapsulating all details except external inputs and outputs. The creating, composing and decomposing of corelets is done in an object-oriented Corelet Language in Matlab.

While the TrueNorth chip is a flexible platform, it does pose several constraints. To maintain

extremely low cost and high energy efficiency, each column in the crossbar only supports 4 different synaptic weights [66] and all the synaptic weights in the crossbar are associated to axon types which are shared by all other neurons of the core. Hence all neurons using a row are required to use the same weight rank i.e. the crossbar supports 256x256 weights but can have only 4x256 unique weights. Also because of the 256x256 crossbar, the fan-in and fan-out per neuron is limited to only 256. These constraints limit the direct mapping from a given SNN to its TrueNorth implementation. To the best of our knowledge, there has not been any public domain tool that converts an arbitrary SNN to the TrueNorth implementation. Though, several applications have been developed on TrueNorth by following design approaches, "train-then-constrain" [67] [68] or "constrain-then-train" [66], which include the methods of constructing and training the network on libraries such as Pylearn2/Theano or Caffe and mapping them onto TrueNorth as per their network.

IBM's TrueNorth processor is very low-power, highly scalable, and optimized for large-scale computing [66]. However, harnessing the strengths of TrueNorth demands algorithms which are adept to its constraints. Recent developments suggests an emergence of neuromorphic adaptations of machine learning algorithms. It has been shown that a "train-and-constrain" approach can be taken to map a Recurrent Neural Network (RNN) based natural language processing task (question classification) to a TrueNorth chip [67] by matching artificial neuron's responses with those of spiking neurons with promising results (74% question classification accuracy, less than 0.025% of cores used and an estimated power consumption of ≈17μW). The same "train-and-constrain" approach is used to map a Deep Neural Network (DNN) on to a TrueNorth chip [68] for a sentiment analysis task. Here, the mapping is possible through substitution of the ReLU neurons in the DNN with integrate-and-fire neurons and adjusting their

neuron thresholds and discretizing the weights using a quantization strategy. Few recognition tasks have also been implemented in other promising neuromorphic hardwares [69] [70]. In this work we also take a "train-and-constrain" approach to implement inference-based Bayesian spiking neural networks on the TrueNorth chip.

In this chapter, we aim at implementing a trained probabilistic inference network which represents a probabilistic graphical model on TrueNorth. It involves two steps: at first the inference network is transformed into a stochastic SNN; and secondly the stochastic SNN is converted into a TrueNorth implementation. Using inference-based sentence construction as a case study, we discuss algorithms that transform an inference network to a spiking neural network, and a spiking neural network to TrueNorth corelet designs. In our experiments, the TrueNorth spiking neural network constructed sentences have a matching accuracy of 88% while consuming an average power of 0.205 mW.

7.1 Normalized Winner-Take-All

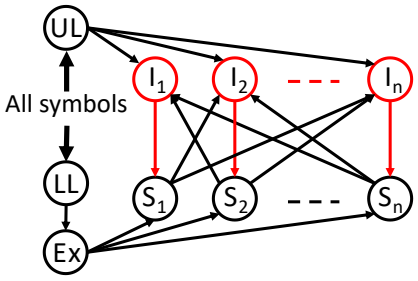


Fig. 47. Normalized winner-take-all NW

The original inference model described in the confabulation process requires that the belief value of all symbols in each lexicon must add up to 1. In a stochastic SNN, this means the total firing activities of neurons in each lexicon must be approximately the same. To achieve this, we introduce *normalized winner-take-all (NWTA)* network. Three neurons, *upper limiter* (*UL*), *lower*

limiter (*LL*), and *exciter* (*Ex*), are added to the previously discussed WTA circuit as shown in Fig. 47. Both the UL and LL are regular integrate and fire neurons, which have input links from all symbol neurons (these links are omitted in the figure for the sake of simplicity). The links to UL have positive weights while the links to LL have negative weights. On the other hand, the UL has negative leakage and LL has positive leakage. The leak values are adjusted in proportion to the number of symbols in the NWTA network. The threshold of these neurons is adjusted for a desired range of firing rates for the network. With this configuration, the UL neuron builds up membrane potential every time it receives spikes from symbols which leak away at a constant rate. If the symbols are firing at a higher rate than the rate of leak, then the UL neuron fires indicating the symbols need to be inhibited. The UL neuron drives all the inhibitors with equal weights, hence suppressing all the symbol neurons equally without disturbing their relative excitation levels. On the other hand, the LL neuron builds up the membrane potential due to leak. The membrane potential drops only if the symbol neurons fire. If this firing rate is lower than required, then LL neuron fires indicating that the symbol neurons are firing at a lower rate than desired. One Ex neuron, which is driven by the LL neuron, provides equal excitation for all the symbol neurons. Similar to the inhibitor neurons the exciter neuron is also of type ReLU and it spreads amplitude of excitation over time, again without disturbing the relative excitation levels of the symbol neurons. Hence this recurrent behavior obtained from the controlled feedback through inhibition and excitation forces the symbol neurons to cumulatively fire in the desired rate-range.

7.2 Overall Network Creation

To build the overall stochastic SNN, each lexicon in the original inference model is

implemented using the above mentioned NWTA network. Excitatory links that connect symbol neurons are established across different lexicons. This network will be referred to as *reference network* in the rest of the paper. Another network will be derived from it and be mapped to the TrueNorth processor, which will be discussed in the next section.

As a case study, we built a small scale SNN based on the trained weights extracted from the intelligent text recognition system (ITRS) [2]. ITRS is trained on a huge corpus of English text. Its knowledge base consists of conditional probabilities between neighboring words and phrases. The number of unique symbols in the knowledge base is about 974,000, which includes words and phrases. Based on this knowledge, it forms anticipations of the word at sentence level context. Given an observation consisting of a fuzzy list of likely word candidates at each word position, familiar information with high relevancy will be recalled resulting in a meaningful and semantically correct sentence. This model has an overall sentence accuracy of 94%.

We extract details only pertaining to few example sentence images to build simple inference networks for evaluating the feasibility of its implementation in stochastic SNN. Each symbol is a potential word or phrase at every word or phrase position of the sentence. All symbol neurons are initialized with the same intrinsic potential (i.e. the same initial firing rate). The most strongly connected neurons resonate and enhance each other across lexicons and at the same time inhibit other neurons in the same lexicon. When network activity settles, neurons with the highest firing rate in each lexicon marks the sentence that is grammatically correct and semantically meaningful.

We randomly picked a set of sentences from document images. Fuzzy character recognition is performed on these images which results in multiple possible words for each word position as described in [2]. For example, given input candidates [{he, ho, be, bo, re, ro, ne, no} {soon}

{found, round} {the, kho, une, unc} {place, placed}], the SNN settles at a grammatically correct sentence as [he soon found the place]. Fig. 48 shows the raster plot for one of the sentences. The labels along Y-axis are grouped by lexicon in the following format; the lexicon number, symbol represented by the neuron and the 3D (x y z) co-ordinate of the neuron in the reference network. The spike count for winning symbols is highest in the Lexicon, which is shown along the secondary Y-axis. The X-axis represents the simulation ticks.

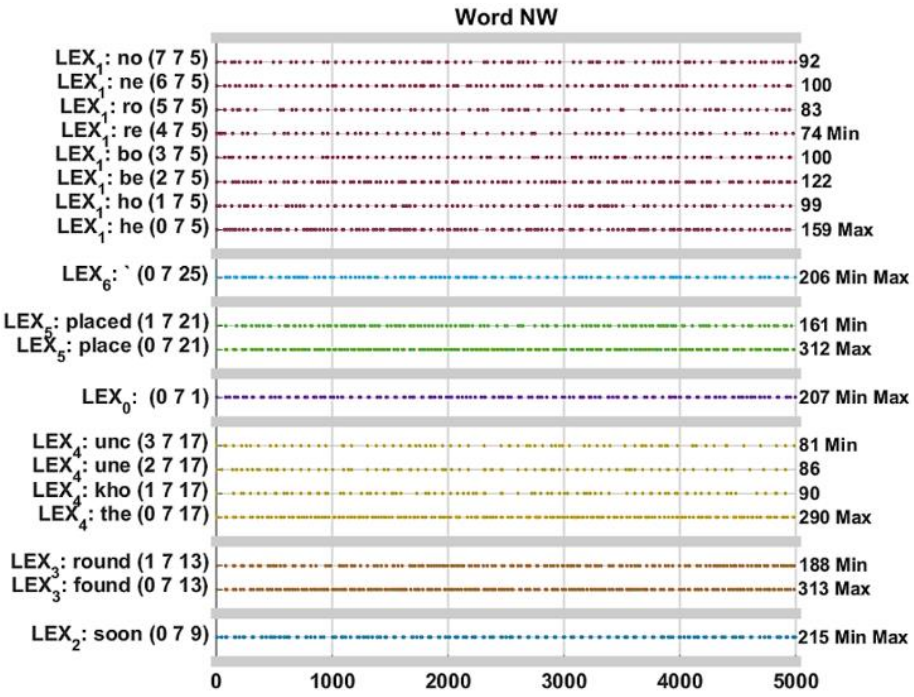


Fig. 48. Reference network results

7.3 Background of TrueNorth Neurosynaptic Processor

TrueNorth is a neurosynaptic processor created by IBM. It is based on the brain's parallel processing architecture and is highly efficient, scalable and flexible. It implements general purpose programmable spiking neurons. The digital architecture of a TrueNorth chip consists

4096 cores [65] each with 256 neurons and 256 axons connected via 256x256 directed synaptic connections, thus providing 1 million programmable neurons and 268 million configurable synapses. TrueNorth uses an efficient event-driven architecture. Address event representation (AER) is adopted for spike representation and communication between neurons. These spike events are sparse in time and active power is proportional to firing activity thus making it highly efficient. The core architecture is as shown in Fig. 49.

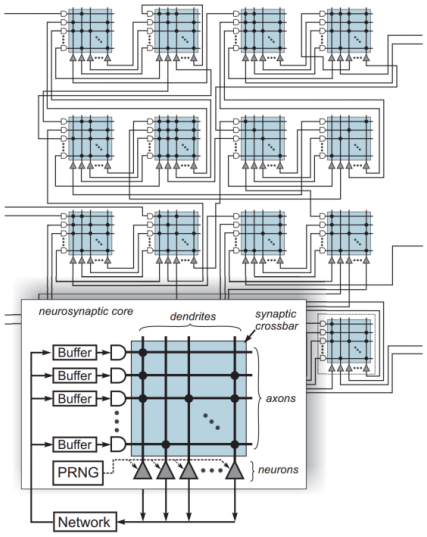


Fig. 49. TrueNorth core fabric

Each neuron is independently configurable with a wide variety of neuron models including stochastic ones. Corelets are used as design language for creating networks. Using the Corelet Programming Environment (CPE) these corelets can be programmed to the chip and evaluated or can be simulated using their 1:1 hardware simulator called Compass [71]. The corelet environment is shown in Fig. 50.

The TrueNorth chip is a low power platform. It uses low leakage transistors for minimizing passive power consumption. Active power is minimized due to the event-driven architecture of the logic where computation is performed only when required [72].

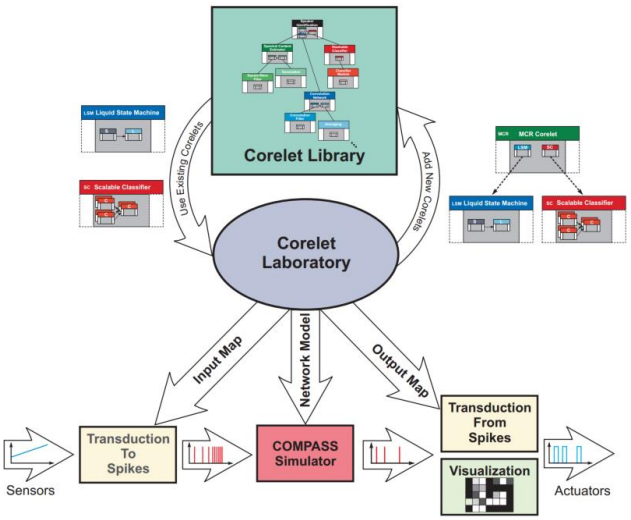


Fig. 50. Corelet programming environment

7.4 Design Flow

Our second step is to transform the reference network to the TrueNorth implementation. This involves 3 more steps as shown in Fig. 51. Based on the reference network, whose construction is described in the previous section, corresponding shadow networks are created, which comply with the physical constraints posed by the TrueNorth hardware. The shadow network is further flattened to corelets where corelet level details are added. The flattening process results in one corelet per lexicon. These corelets are now connected with each other to build the physical network. The connected corelets are finally simulated using the compass simulator and the TrueNorth chip is programmed for evaluation in real-time.

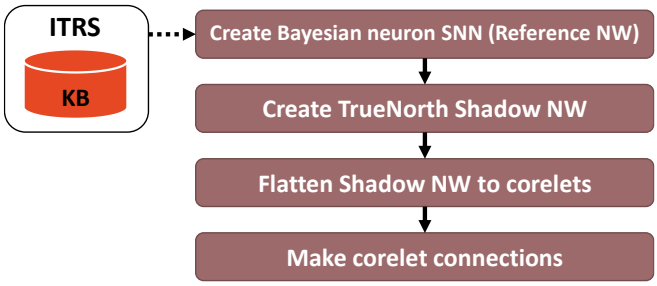
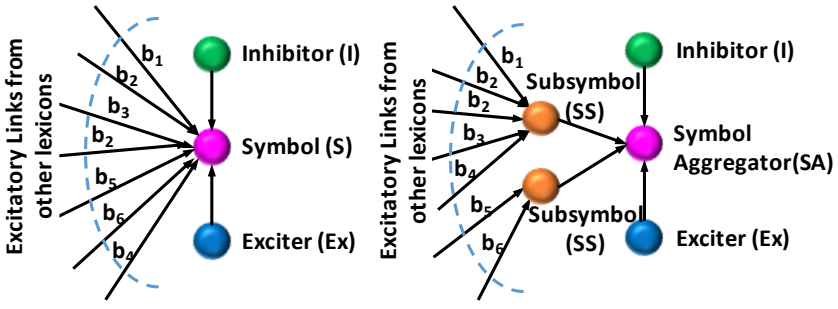


Fig. 51. Design flow

7.5 SHADOW NETWORK CREATION

For every reference network, we generate an equivalent TrueNorth compatible shadow network. This network complies with the restrictions imposed by the platform. Due to the hardware restrictions of TrueNorth, some simplifications of the Bayesian neuron must be adopted. For TrueNorth implementation, we replace the Bayesian neuron with a stochastic integrate and fire neuron to obtain similar characteristics. From the model described above we infer that there are two computational stages which are unique to the Bayesian model in contrast to the regular integrate and fire neuron model. The first being the exponent function and the other being the Poisson firing. We suggest skipping the exponent computation and using the accumulation of weighted spikes to directly compute the membrane potential. The Bayesian neuron must operate over a small region of the exponential to maintain its dynamic range. This can be approximated with a linear function, which is inherent to the accumulation of membrane potential in an integrate and fire neuron. For Poisson spike generation, we suggest randomly varying the threshold after every spike generation. The Bayesian neuron's output firing rate is exponentially proportional to the cumulative rate of input weighted spikes. By limiting the range of threshold change to a small interval which satisfies the exponential distribution with unit rate, we achieve a firing pattern similar to Poisson spiking behavior as described in the model. The

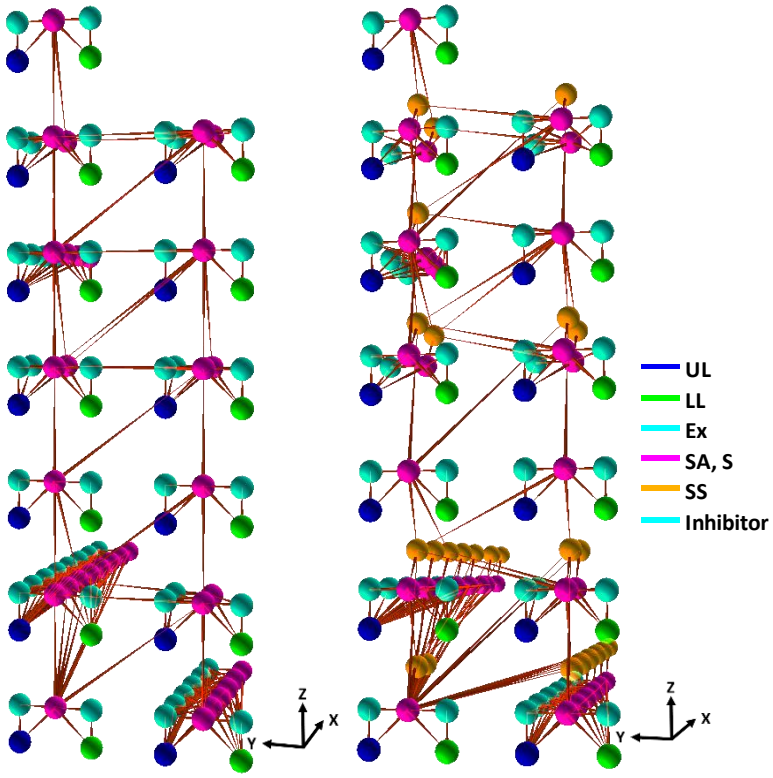


(a) Reference network (b) Shadow network

Fig. 52. Comparing reference network and shadow network

general behavior of neuron is still similar to the Bayesian neuron model even with these simplifications. The TrueNorth neuron is configured with this behavior for symbol neurons to obtain a stochastic neuron. The rest of the neurons used in the network can be directly configured to TrueNorth neurons.

Another hardware limitation of TrueNorth is that, although a neuron may receive connections from many axons, the weights of these synaptic links can have only a combination of four different values. Consider a symbol neuron in Fig. 52 (a), it has connections from the inhibitor, the exciter and numerous excitatory links from symbol neurons in other lexicons. All these links have different weights. Hence it cannot be directly implemented as a TrueNorth neuron. Our solution is to decompose a symbol neuron into multiple *sub-symbol* (*SS*) neurons and a *symbol aggregator* (*SA*). For each symbol neuron in the reference network, first, we scale the weight of



(a) (b)

Fig. 53. SpNSim 3D NW visualization a) Reference NW with Bayesian neurons b) TrueNorth equivalent shadow NW

its incoming links to an integer range. After scaling, these weights are binned for quantization, based on user specified bin width. Different binning strategies can be used. All links falling into the same bin will be assigned the same weight, which is the rounded average of their original weight. For every 4 bins, a sub-symbol neuron is created as the receptor of all links falling into these bins. The sub-symbol neurons are of type ReLU and connect to a symbol aggregator neuron via link with unit weight. Hence, their function is to collect and accumulate input spikes and relay the results to the aggregator over time. The symbol aggregator is a stochastic integrate and fire neuron as previously discussed. It aggregates inputs and generates spikes that will be sent to inhibitors, UL, LL neurons, as well as sub-symbol neurons in other lexicons.

An example of SA and SS neurons is given in Fig. 52 (b). In the figure, the symbol in the reference network has 7 incoming connections falling into 6 bins. In the shadow network, 2 SS neurons are created. The first one receives 5 incoming connections distributed over 4 different bins, while the second one receives the rest of the incoming links originally connecting to the symbol neuron. Both SS neurons connect to an SA neuron, which also receives input from the inhibitor and exciter. Since all SS neurons connect to the SA neuron with unit weight, there is no limitation of the number of SS neurons that can be created, as long as there are enough hardware neuron resources. The above procedure is not necessary for neurons other than the symbol neuron, i.e. inhibitor, Ex, LL and UL neurons. They can directly be implemented as a TrueNorth neuron, because their incoming links have the same weight, in other words, they all fall into the same bin. The visualizations for the case of reference network and its equivalent TrueNorth compatible shadow network is shown in Fig. 53. Different type of neurons are shown in different colors as illustrated in the legend. Compared to the reference network in Fig. 53. (a), the shadow network in Fig. 53. (b) has the added SS neurons (in orange) and their links. These networks are

for the same sentence as shown in Fig. 48. The left and right sub-network represent the word and phrase lexicons respectively. The symbols in each lexicon are shown along X-axis.

7.6 Flattening the Shadow Network

We have developed a parameterized lexicon corelet in CPE. The parameters are the number of symbols, their associated weights, input connectivity and output connectivity. The corelet has one input connector that receives incoming signals from other corelets and two output connectors, one of them is the internal connector that sends the output to the input connector of other corelets; and the other is an external connector that sends the output to the I/O pins. In the TrueNorth architecture, a neuron can only drive a single axon which has a fan out of 256 within a single

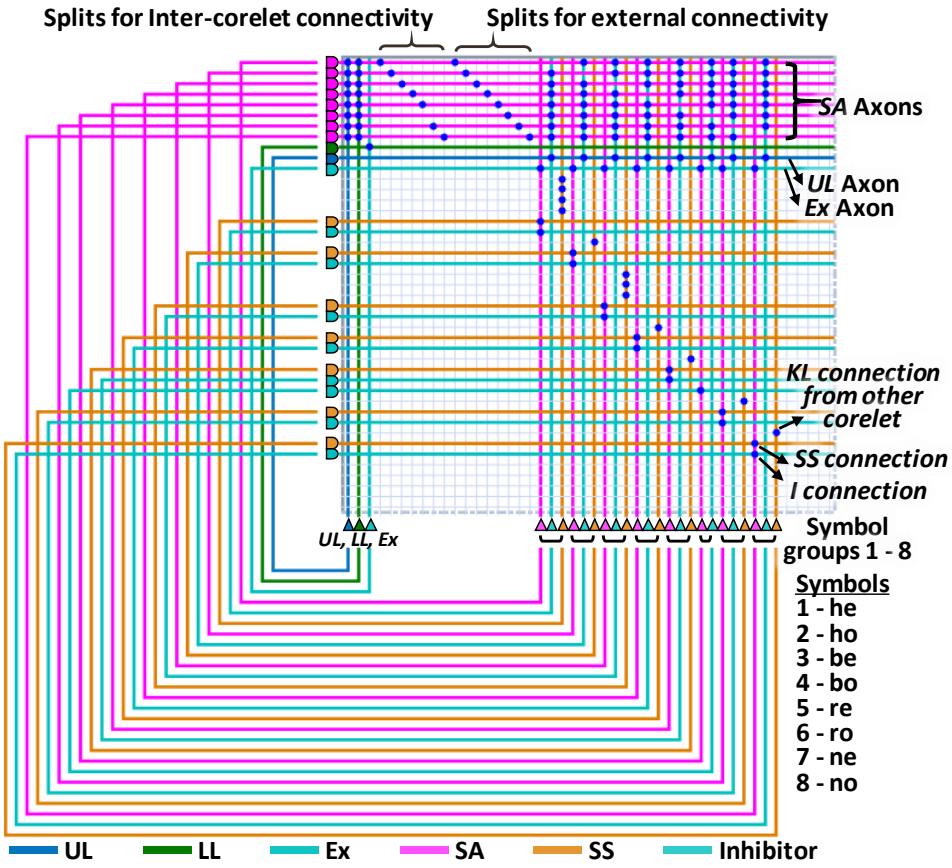


Fig. 54. Crossbar connections for lexicon 1

core. In cases where a neuron must send spikes to downstream neurons in multiple cores then the upstream neuron must undergo splitting. To achieve this the splitting neuron is configured with weight equal to 1 and threshold equal to 1.

Fig. 54 shows a flattened network for lexicon 1 of the example sentence given in Fig. 48, which is the crossbar of a core. The rows of the crossbar are the axons and the columns are dendrites. The neurons are placed along the bottom of the crossbar. A dot in the crossbar represents a synapse. The figure shows wiring for only loopback connections. The dendrites, axons, loopback connections and the neurons are depicted based on the color legend given in the figure. From the crossbar, we can see that UL (i.e. dark blue) and LL (i.e. green) neurons have synapses from all magenta axons looped back from SA neurons. Those magenta colored axons also generate splits for inter-corelet and external connectivity, and connect to all inhibitor (i.e. cyan) neurons. Most SA (i.e. magenta colored) neurons have 3 synapses, coming from orange (i.e. SS), blue (i.e. Ex) and cyan (i.e. inhibitor) axons. All SS (i.e. orange colored) neurons have synapses from axons that are not colored, which means links from other corelets (i.e. lexicons). With these connections, we flatten the NWTA network. The pseudo code to setup cores and neurons is given below. For convenience, the SA, I and SS neurons associated with a symbol is referred to as symbol group (SG).

The core and neuron setup process involves making crossbar connections and configuring neurons. The first step in this process as shown in the pseudo code is to determine the number of axons which must be reserved in a core. These axons will be shared among all neurons of the core. Then a neuron list is prepared which involves all neurons in the lexicon including the splits for connectors. Neurons from this list are sequentially added to cores and appropriate synapses are configured. If a neuron cannot be accommodated due to a lack of enough neurons for making

splits or due to not enough axons for all other neuron types, then an additional core is added and the process continues. Once all the neurons are assigned then the number of splits required to support the reserved axons in each core is computed and these additional inter-core split neurons are assigned using the same method as described above. The resulting core allocation is as shown in Fig. 55. Hence the number of cores and neurons instantiated are based on the network topology.

After the core and neuron configuration is done the connectivity inside cores is now established. This involves linking the neuron outputs to axons. The connectivity between input connector and axons is established followed by connectivity between neurons and output connectors. Any unused neurons in all cores are configured to be disconnected. This completes the corelet creation process, which is repeated till all the lexicons are flattened. In this way, we map the network in a compact manor to TrueNorth.

7.7 Creating Connected Corelets

After creating all the corelets they are connected based on the connectivity information between lexicons which represents KLs, resulting in a fully formed stochastic SNN. This list of corelets is compiled and run on both the compass simulator and on the TrueNorth chip.

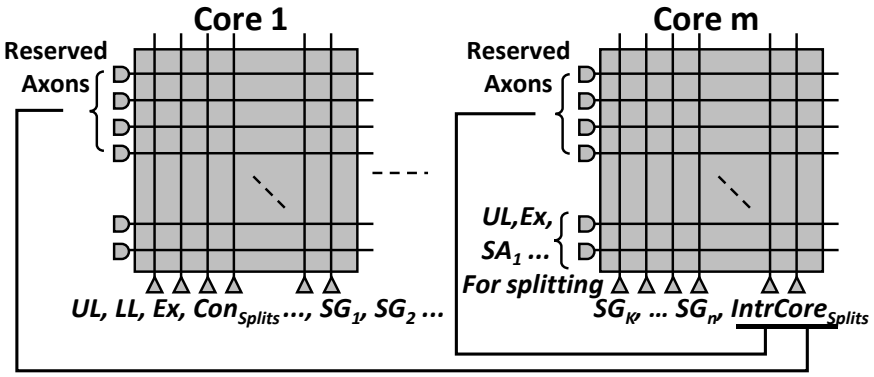


Fig. 55. Core and neuron allocation

The example sentence utilizes 13 cores with the above algorithm. In general, the number of corelets generated will be equal to the number of lexicons in the sentence and each corelet will use as many cores required to map all the symbols in the lexicon.

Algorithm 5. Setup Cores and Neurons

```
numReserveAxons ← (2 + number of SA neurons)
NumLst ← All neurons in lexicon //UL,LL,Ex,Cons_Splits,SG_1-n
NumLst_itr ← begin(NumLst) // initialize iterator
splitsDone ← false
done ← false
while not done
   ADD Core
   RESERVE axons for SA, UL, LL and EX in the core
   for i ← 1 to 256
      curNum ← NumLst(NumLst_itr)
      DETERMINE resources required, numNeurons for splits, numAxons
      for all other neurons
      if resource available in core then
         ASSIGN neuron type
         if (curNum = UL or LL) then
            CREATE crossbar connection from the SA axon to UL and LL
         else if (curNum = Ex) then
            CREATE crossbar connection from the LL axon to Ex
         else if(curNum = SS) then
            CREATE crossbar connection from the corelet input axons to SS
         else if(curNum = SA) then
            CREATE crossbar connection from SS, I, Ex to SA
         else if(curNum = I) then
            CREATE crossbar connection from SA, UL to I
         else if(curNum = split) then
            CREATE crossbar connections from given axon.
         INCREMENT NumLst_itr
         if (end(NumLst) = NumLst_itr) then //check for end of list
            if splitsDone then
               done ← true
               Break
            else
               splitsDone ← true
               COMPUTE number of splits required for UL, Ex and SA
               neurons to support number of cores created
               NumLst ←split neurons to feed reserved axons
               NumLst_itr ← begin(NumLst) // initialize iterator
      else
Break
```

7.8 DESIGN ENVIRONMENT

In this section we briefly describe the tools used and the design environment details which is as shown in Fig. 56. The input of the design environment is the confabulation model and the trained knowledge base, which provides lexicon details and knowledge link information. The network creator generates the reference network and the shadow network.

The reference network is sent to an in-house spiking neural network simulator, SpNSim for

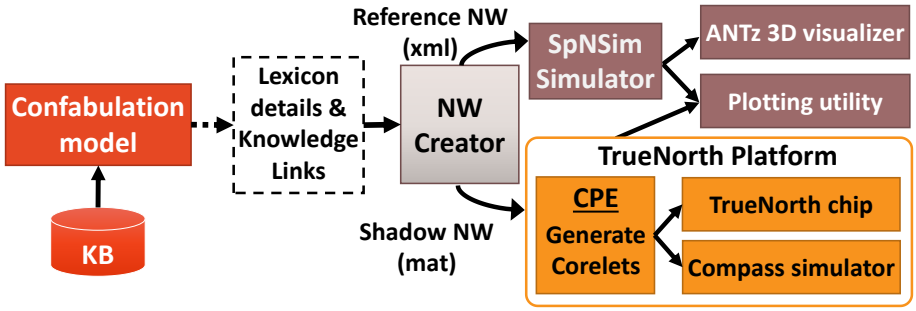


Fig. 56. Design Environment

functional verification [73]. SpNSim is a multithreaded and scalable simulation platform built using C++. It is flexible and vectorized for efficient evaluation and capable of handling large-scale networks of mixed neuron models. It is also capable of training stochastic SNNs using STDP. In SpNSim, the network topology along with the network parameters are specified through a XML input file. The XML input file also includes definition of runtime parameters to specify the dynamic behavior (e.g. the starting and ending of learning phase, test phase, etc.) of the network. SpNSim is capable of generating visualization data which is used to render 3D neural networks using an open source 3D data visualization tool called ANTz. This is a very helpful feature for debugging and analyzing complex neural networks. There is an accompanying plotting utility developed in MATLAB for plotting and analyzing spikes. It is also capable of analyzing neuron parameters and visualizing weight evolution for debug purpose.

The tools that convert the reference network to the shadow network are developed in C++. The shadow network is sent to CPE where it is flattened and corelets are generated. Since CPE is a MATLAB based tool, the shadow network is saved as a MAT file, which contains the neuron details, weights, connectivity and network topology information required for creating corelets. Finally, the connected corelets are simulated using the IBM compass simulator or executed on the TrueNorth processor. After evaluating the neural network, we convert the TrueNorth spike files to SpNSim spike file format and create raster plots and signal to noise ratio (SNR) plots using the

plotting utility of SpNSim.

7.9 EXPERIMENTS AND RESULTS

Experiments have been carried out to validate the functionality of the TrueNorth implementations generated from the above design flow. Multiple networks are built, each capable of confabulating one sentence from a set of possible words. These networks do not have inputs, the stochastic firing gets amplified due to resonance and produce meaningful sentences as outputs.

A random set of 100 sentences from "Ali Baba and Forty Thieves" were picked from noisy document images as our test cases. This text was not used for training. The reference networks were simulated using SpNSim and the results are compared to the output of the SNN running on TrueNorth. For 85% of the test cases, TrueNorth generated the same sentence as the simulated reference network with a sentence accuracy of 70%. Fig. 57 shows the raster plot for results

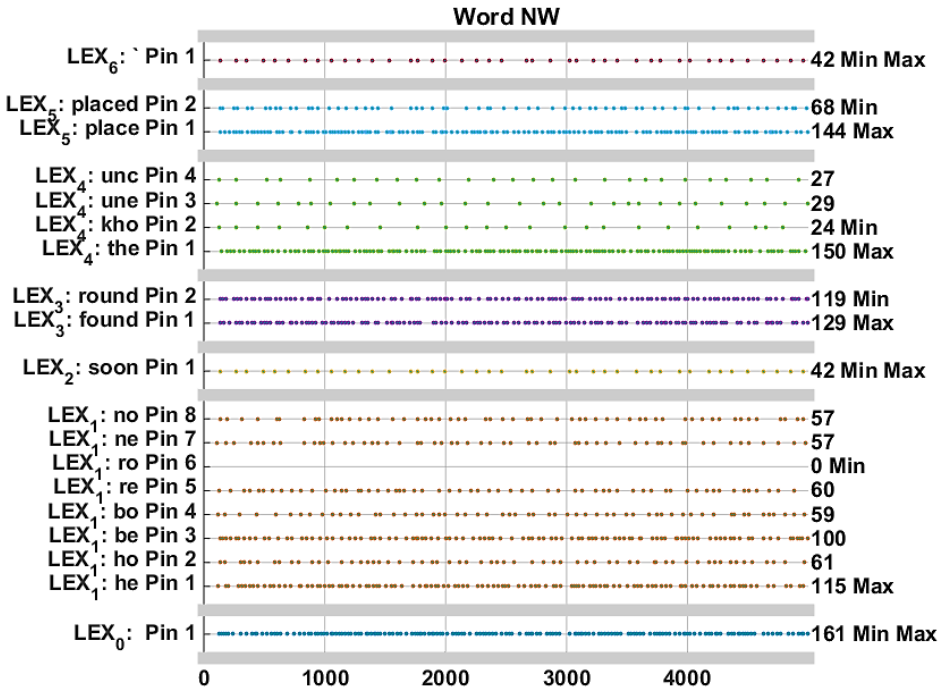


Fig. 57. TrueNorth sentence results

obtained from TrueNorth. The labels and axes represent the same information as in Fig. 48 except that the Y-axis labels show the corelet connector pin numbers.

There is no timing control on the neurons to reset its operation over window intervals. These are free running neurons that amplify or suppress the excitation inherently. Therefore, window analysis is performed as post processing on the collected spikes to determine a feasible window over which the spikes must be accumulated to get statistically significant results. It is important to note that, these are resonant networks without external stimulus resulting in sparse spiking patterns. When stimulated by external inputs, corresponding neurons and their downstream neighbors will increase the spiking density. Also, the spiking density can be tuned by varying the thresholds of UL and LL neurons, based on the application requirement. For window analysis, the number of spikes is counted for each window and plotted. The gap between the desired curve and rest of the curves represents the signal to noise ratio (SNR). The larger the gap, the higher is the SNR for detection. The effects of sampling the output for different window sizes is shown in Fig. 58 for lexicon 1. From the figure, it is evident that a window size of about 500 ticks is enough to reliably determine the confabulated result, which is used in all the experiments.

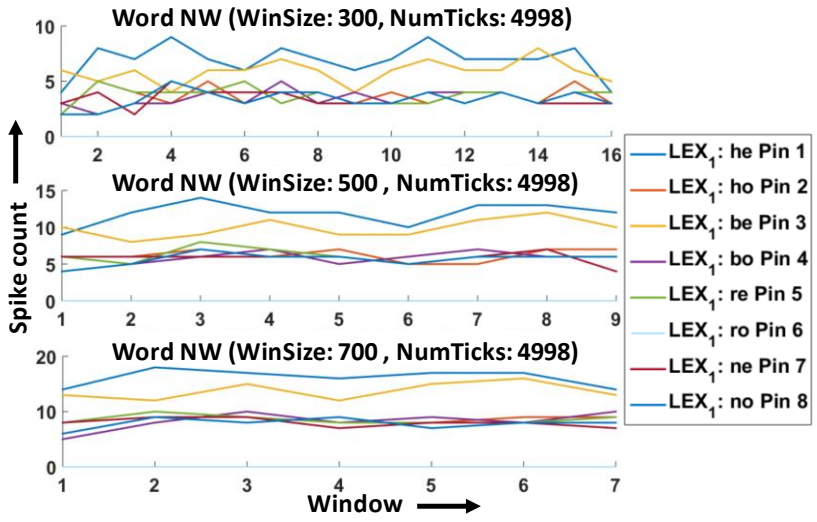


Fig. 58. Effect of window size on lexicon 1

While making the network compatible to TrueNorth, we had to scale and quantize the weights. By increasing the bin width, more connections will share the same weight and will result in lower performance. This effect is shown in Fig. 59 for different bin widths. Since these networks are small the effect is small but the trend is visible. As a rule of thumb, reducing bin width increases precision but at the cost of additional resources.

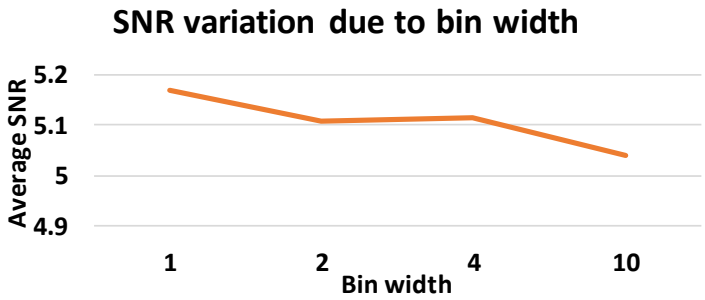


Fig. 59. Effect of bin width on SNR.

After programming the TrueNorth chip we measure the power consumption to run the network as characterized in [72]. To find the actual power consumption just for the resources utilized on the chip, first the leakage power P_{leak} is measured when the system is idle then another power measurement P_{total} is made with the network running. The active power is computed as $P_{active} = P_{total} - P_{leak}$. The leakage power is scaled to get the leakage power for only the cores utilized out of 4096 cores, $P_{leak\_s} = P_{leak} * \text{NumCores} / 4096$. Finally, the total scaled power is computed as $P_{total\_s} = P_{active} + P_{leak\_s}$. Power measurements were made for four sentences, each for all the four bin widths. Hence 16 networks were used to measure power. As presented in [74], on average the total scaled power is 0.205mW when running the chip at 0.8V and 0.442mW when running the chip at 1.0V.

8 CONCLUSION

Through this work a comprehensive approach is presented to address the issue of effectively designing complex neuromorphic applications which have heavy requirements for computing large amounts of data. We have presented a HPC design methodology for complex data dependent neuromorphic applications for heterogeneous clusters involving data strong dependencies with variety of different workload processing requirements. A scalable architecture is presented to implement such applications as pipelined and distributed systems on a heterogeneous cluster. We also proposed a structure based scheduling scheme to enable seamless scaling and provide module level load balancing in a non-centralized way. Hence achieving maximum resource utilization and providing best throughput for available hardware resources. The proposed architecture is efficient as it performs computation in out-of-order fashion through asynchronous pipelines. Resource mapping algorithms are also presented to efficiently map such complex pipelines to any given heterogeneous cluster for achieving best possible throughput. This is validated by implementing a neuromorphic application and every aspect of the framework is demonstrated. Resource mapping results are compared against existing implementations to show the soundness of results.

For efficient and low power implementation of brain inspired computing applications an entire framework is developed using biologically inspired SNN. Spiking neuron models were presented to enable development of large scale SNNs capable of distributed online learning. To enable such research, a flexible, scalable and high performance simulation platform SpNSim is developed and its architecture is presented. The simulator has the ability to model biologically inspired but non-biologically realistic neuron models which enable efficient computation. We also demonstrate the

functionality of the simulator for learning through STDP and evaluation of inference networks. These network results were validated based on other existing platforms.

A general-purpose, efficient and scalable Bayesian neuron model along with a digital logic design for pipelined implementation which is capable of in-hardware learning is proposed. The proposed model is simulated and validated using two different SNNs applications and compared with existing implementations.

This work also demonstrated that a stochastic integrate and fire neuron model can be used instead of more complex Bayesian neuron model for inference related tasks in spiking neural networks. To achieve this, we have proposed a normalized winner-take-all network topology. We have implemented several examples to verify that both kinds of networks, ones with Bayesian neuron model and another with a stochastic integrate and fire neuron model which produce similar results. The stochastic integrate and fire neuron model based network has been successfully programmed to the IBM TrueNorth neurosynaptic chip and evaluated the network in real-time. We have shown that Bayesian inference computation can be performed in very low power and efficient manner by performing a sentence confabulation task using spiking neural networks.

9 REFERENCES

[1] R. Wray, C. Lebiere, P. Weinstein, K. Jha, J. Springer, T. Belding, B. Best and V. Parunak, "Towards a complete, multi-level cognitive architecture," in *Proc. of the International Conference for Cognitive Modeling*, 2007.

[2] Q. Qiu, Q. Wu, M. Bishop, R. E. Pino and R. W. Linderman, "A Parallel Neuromorphic Text Recognition System and Its Implementation on a Heterogeneous High-Performance Computing Cluster," *IEEE Transactions on Computers,* vol. 62, pp. 886-899, May 2013.

[3] R. Ananthanarayanan, S. K. Esser, H. D. Simon and D. S. Modha, "The cat is out of the bag: cortical simulations with 109 neurons, 1013 synapses," in *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, 2009.

[4] A. Gupta and L. N. Long, "Character recognition using spiking neural networks," in *Neural Networks, 2007. IJCNN 2007. International Joint Conference on*, 2007.

[5] T. Behi, N. Arous and N. Ellouze, "Self-organization map of spiking neurons evaluation in phoneme classification," in *Sciences of Electronics, Technologies of Information and Telecommunications (SETIT), 2012 6th International Conference on*, 2012.

[6] Y. Wang, T. Tang, L. Xia, B. Li, P. Gu, H. Yang, H. Li and Y. Xie, "Energy efficient RRAM spiking neural network for real time classification," in *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, 2015.

[7] P. OĆonnor, D. Neil, S.-C. Liu, T. Delbruck and M. Pfeiffer, "Real-time classification and sensor fusion with a spiking deep belief network," *Frontiers in neuroscience,* vol. 7, 2013.

[8] M. Beyeler, N. D. Dutt and J. L. Krichmar, "Categorization and decision-making in a neurobiologically plausible spiking network using a STDP-like learning rule," *Neural Networks,* vol. 48, pp. 109-124, 2013.

[9] W. Maass, "Networks of spiking neurons: the third generation of neural network models," *Neural networks,* vol. 10, pp. 1659-1671, 1997.

[10] J. Sjöström and W. Gerstner, "Spike-timing dependent plasticity," *Spike-timing dependent plasticity,* vol. 35, 2010.

[11] M. W. Oram, M. C. Wiener, R. Lestienne and B. J. Richmond, "Stochastic nature of precisely timed spike patterns in visual system neuronal responses," *Journal of neurophysiology,* vol. 81, pp. 3021-3033, 1999.

[12] H. S. Seung, "Learning in spiking neural networks by reinforcement of stochastic synaptic transmission," *Neuron,* vol. 40, pp. 1063-1073, 2003.

[13] T. A. Engel, W. Chaisangmongkon, D. J. Freedman and X.-J. Wang, "Choice-correlated activity fluctuations underlie learning of neuronal category representation," *Nature Communications,* vol. 6, 2015.

[14] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica and others, "A view of cloud computing," *Communications of the ACM,* vol. 53, pp. 50-58, 2010.

[15] C. Gong, J. Liu, Q. Zhang, H. Chen and Z. Gong, "The Characteristics of Cloud Computing," in *2010 39th International Conference on Parallel Processing Workshops*, 2010.

[16] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman and J. Good, "On the use of cloud computing for scientific workflows," in *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, 2008.

[17] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM,* vol. 51, pp. 107-113, 2008.

[18] L. Wang, J. Tao, R. Ranjan, H. Marten, A. Streit, J. Chen and D. Chen, "G-Hadoop: MapReduce across distributed data centers for data-intensive computing," *Future Generation Computer Systems,* vol. 29, pp. 739-750, 2013.

[19] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight and others, "Merrimac: Supercomputing with streams," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003.

[20] G. Z. Sun and G. Chen, "Distributed Pipeline Programming Framework for State-Based Pattern," in *2009 Eighth International Conference on Grid and Cooperative Computing*, 2009.

[21] S. Yamagiwa, L. Sousa and T. Brandao, "Meta-Pipeline: A New Execution Mechanism for Distributed Pipeline Processing," in *Parallel and Distributed Computing, 2007. ISPDC '07. Sixth International Symposium on*, 2007.

[22] L. Schor, A. Tretter, T. Scherer and L. Thiele, "Exploiting the parallelism of heterogeneous systems using dataflow graphs on top of OpenCL," in *The 11th IEEE Symposium on Embedded Systems for Real-time Multimedia*, 2013.

[23] S. G. Ahmad, C. S. Liew, M. M. Rafique, E. U. Munir and S. U. Khan, "Data-Intensive Workflow Optimization Based on Application Task Graph Partitioning in Heterogeneous Computing Systems," in *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*, 2014.

[24] K. Agrawal, A. Benoit and Y. Robert, "Mapping linear workflows with computation/communication overlap," in *Parallel and Distributed Systems, 2008. ICPADS'08. 14th IEEE International Conference on*, 2008.

[25] Q. Wu, Y. Gu, M. Zhu and N. S. V. Rao, "Optimizing network performance of computing pipelines in distributed environments," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008.

[26] M. Zhu, Q. Wu, N. S. V. Rao and S. S. Iyengar, "On optimal mapping of visualization pipeline onto linear arrangement of network nodes," in *Electronic Imaging 2005*, 2005.

[27] A. Benoit, J.-M. Nicod and V. Rehn-Sonigo, "Optimizing buffer sizes for pipeline workflow scheduling with setup times," in *Parallel \& Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, 2014.

[28] H. Singh and R. Randhawa, "Cuckoo search based workflow scheduling on heterogeneous cloud resources," in *Cloud Computing, Data Science \& Engineering-Confluence, 2017 7th International Conference on*, 2017.

[29] E. E. Mon, M. M. Thein and M. T. Aung, "Clustering Based on Task Dependency for Data-Intensive Workflow Scheduling Optimization," in *2016 9th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers (MTAGS)*, 2016.

[30] K. Wang, K. Qiao, I. Sadooghi, X. Zhou, T. Li, M. Lang and I. Raicu, "Load-balanced and locality-aware scheduling for data-intensive workloads at extreme scales," *Concurrency and Computation: Practice and Experience,* vol. 28, pp. 70-94, 2016.

[31] M. Wang, J. Zhang, F. Dong and J. Luo, "Data Placement and Task Scheduling Optimization for Data Intensive Scientific Workflow in Multiple Data Centers Environment," in *2014 Second International Conference on Advanced Cloud and Big Data*, 2014.

[32] J. Park and Y. Park, "An optimization approach to design of generalized BSB neural associative memories," *Neural computation,* vol. 12, pp. 1449-1462, 2000.

[33] Y. Park, "Optimal and robust design of brain-state-in-a-box neural associative memories," *Neural Networks,* vol. 23, pp. 210-218, 2010.

[34] A. Schultz, "Collective recall via the Brain-State-in-a-Box network," *Neural Networks, IEEE Transactions on,* vol. 4, pp. 580-587, 1993.

[35] R. Hecht-Nielsen, Confabulation theory: the mechanism of thought, Springer Heidelberg, 2007.

[36] K. Ahmed, Q. Qiu and M. Tamhankar, "Distributed and configurable architecture for neuromorphic applications on heterogeneous cluster," in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, 2016.

[37] K. Ahmed, Q. Qiu, P. Malani and M. Tamhankar, "Accelerating pattern matching in neuromorphic text recognition system using intel xeon phi coprocessor," in *Neural Networks (IJCNN), 2014 International Joint Conference on*, 2014.

[38] Z. R. Yang and M. Zwolinski, "Mutual information theory for adaptive mixture models," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 23, pp. 396-403, 2001.

[39] Z. Li, Q. Qiu and M. Tamhankar, "Towards parallel implementation of associative inference for cogent confabulation," in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, 2016.

[40] M. L. Hines and N. T. Carnevale, "The NEURON simulation environment," *NEURON,* vol. 9, 2006.

[41] J. M. Bower and D. Beeman, The book of GENESIS: exploring realistic neural models with the GEneral NEural SImulation System, Springer Science & Business Media, 2012.

[42] B. Nessler, M. Pfeiffer, L. Buesing and W. Maass, "Bayesian computation emerges in generic cortical microcircuits through spike-timing-dependent plasticity," *PLoS computational biology,* vol. 9, p. e1003037, 2013.

[43] *http://white.stanford.edu/pdcwiki/index.php/Spike_generation_using_a_Poisson_process.*

[44] K. Doya, Bayesian brain: Probabilistic approaches to neural coding, MIT press, 2007.

[45] R. P. N. Rao, B. A. Olshausen and M. S. Lewicki, Probabilistic models of the brain: Perception and neural function, MIT press, 2002.

[46] S. Deneve, "Bayesian inference in spiking neurons," in *Advances in neural information processing systems*, 2005.

[47] D. Goodman and R. Brette, "Brian: a simulator for spiking neural networks in Python," *Frontiers in neuroinformatics,* vol. 2, 2008.

[48] M.-O. Gewaltig and M. Diesmann, "Nest (neural simulation tool)," *Scholarpedia,* vol. 2, p. 1430, 2007.

[49] E. Ros, R. Carrillo, E. M. Ortigosa, B. Barbour and R. Agʕis, "Event-driven simulation scheme for spiking neural networks using lookup tables to characterize neuronal dynamics," *Neural computation,* vol. 18, pp. 2959-2993, 2006.

[50] *http://sccn.ucsd.edu/ arno/spikenet/.*

[51] S. B. Furber, D. R. Lester, L. A. Plana, J. D. Garside, E. Painkras, S. Temple and A. D. Brown, "Overview of the spinnaker system architecture," *IEEE Transactions on Computers,* vol. 62, pp. 2454-2467, 2013.

[52] *http://openantz.com/.*

[53] H. Lee, R. Grosse, R. Ranganath and A. Y. Ng, "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations," in *Proceedings of the 26th annual international conference on machine learning*, 2009.

[54] E. Arguello, R. Silva, C. Castillo and M. Huerta, "The neuroid: A novel and simplified neuron-model," in *Engineering in Medicine and Biology Society (EMBC), 2012 Annual International Conference of the IEEE*, 2012.

[55] O. Bichler, D. Querlioz, S. J. Thorpe, J.-P. Bourgoin and C. Gamrat, "Extraction of temporally correlated features from dynamic vision sensors with spike-timing-dependent plasticity," *Neural Networks,* vol. 32, pp. 339-348, 2012.

[56] S. Park, A. Sheri, J. Kim, J. Noh, J. Jang, M. Jeon, B. Lee, B. R. Lee, B. H. Lee and H. Hwang, "Neuromorphic speech systems using advanced ReRAM-based synapse," in *Electron Devices Meeting (IEDM), 2013 IEEE International*, 2013.

[57] J. M. Cruz-Albrecht, M. W. Yung and N. Srinivasa, "Energy-efficient neuron, synapse and STDP integrated circuits," *IEEE transactions on biomedical circuits and systems,* vol. 6, pp. 246-256, 2012.

[58] B. V. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A. R. Chandrasekaran, J.-M. Bussat, R. Alvarez-Icaza, J. V. Arthur, P. A. Merolla and K. Boahen, "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations," *Proceedings of the IEEE,* vol. 102, pp. 699-716, 2014.

[59] J. Schemmel, D. Briiderle, A. Griibl, M. Hock, K. Meier and S. Millner, "A wafer-scale neuromorphic hardware system for large-scale neural modeling," in *Circuits and systems (ISCAS), proceedings of 2010 IEEE international symposium on*, 2010.

[60] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura and others, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science,* vol. 345, pp. 668-673, 2014.

[61] G. Taylor and G. Cox, "Digital randomness," *IEEE spectrum,* vol. 48, pp. 32-58, 2011.

[62] P. U. Diehl and M. Cook, "Unsupervised learning of digit recognition using spike-timing-dependent plasticity," *Frontiers in computational neuroscience,* vol. 9, 2015.

[63] K. Ahmed, A. Shrestha, Y. Wang and Q. Qiu, "System Design for In-Hardware STDP Learning and Spiking Based Probablistic Inference," in *VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on*, 2016.

[64] H. Lee, P. Pham, Y. Largman and A. Y. Ng, "Unsupervised feature learning for audio classification using convolutional deep belief networks," in *Advances in neural information processing systems*, 2009.

[65] A. Amir, P. Datta, W. P. Risk, A. S. Cassidy, J. A. Kusnitz, S. K. Esser, A. Andreopoulos, T. M. Wong, M. Flickner, R. Alvarez-Icaza and others, "Cognitive computing programming paradigm: a corelet language for composing networks of neurosynaptic cores," in *Neural Networks (IJCNN), The 2013 International Joint Conference on*, 2013.

[66] S. K. Esser, A. Andreopoulos, R. Appuswamy, P. Datta, D. Barch, A. Amir, J. Arthur, A. Cassidy, M. Flickner, P. Merolla and others, "Cognitive computing systems: Algorithms and applications for networks of neurosynaptic cores," in *Neural Networks (IJCNN), The 2013 International Joint Conference on*, 2013.

[67] P. U. Diehl, G. Zarrella, A. Cassidy, B. U. Pedroni and E. Neftci, "Conversion of artificial recurrent neural networks to spiking neural networks for low-power neuromorphic hardware," in *Rebooting Computing (ICRC), IEEE International Conference on*, 2016.

[68] P. U. Diehl, B. U. Pedroni, A. Cassidy, P. Merolla, E. Neftci and G. Zarrella, "Truehappiness: Neuromorphic emotion recognition on truenorth," in *Neural Networks (IJCNN), 2016 International Joint Conference on*, 2016.

[69] E. Stromatias, M. Pfeiffer, F. Galluppi, S. B. Furber and S.-C. Liu, "Robustness of spiking deep belief networks to noise and reduced bit precision of neuro-inspired hardware platforms," *Frontiers in neuroscience,* vol. 9, p. 222, 2015.

[70] D. Neil and S.-C. Liu, "Minitaur, an event-driven FPGA-based spiking network accelerator," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 22, pp. 2621-2628, 2014.

[71] R. Preissl, T. M. Wong, P. Datta, M. Flickner, R. Singh, S. K. Esser, W. P. Risk, H. D. Simon and D. S. Modha, "Compass: A scalable simulator for an architecture for cognitive computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[72] A. S. Cassidy, R. Alvarez-Icaza, F. Akopyan, J. Sawada, J. V. Arthur, P. A. Merolla, P. Datta, M. G. Tallada, B. Taba, A. Andreopoulos and others, "Real-time scalable cortical computing at 46 giga-synaptic OPS/watt with," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2014.

[73] K. Ahmed, A. Shrestha and Q. Qiu, "Simulation of bayesian learning and inference on distributed stochastic spiking neural networks," in *Neural Networks (IJCNN), 2016 International Joint Conference on*, 2016.

[74] K. Ahmed, A. Shrestha, Q. Qiu and Q. Wu, "Probabilistic inference using stochastic spiking neural networks on a neurosynaptic processor," in *Neural Networks (IJCNN), 2016 International Joint Conference on*, 2016.

10 VITA

Name Khadeer Ahmed

Education

- **Masters of Science in Computer Engineering,** Syracuse University, Syracuse, NY
- **Bachelor of Engineering in Electronics and Communication,** R.N.S. Institute of Technology, Bangalore, India

Professional Experience

- Research Assistant and Teaching Assistant at **Department of Electrical & Computer Engineering, Syracuse University,** Syracuse, New York, USA
- Intern at **Intel,** Santa Clara, California, USA
- Member Technical – Hardware at **iWave Systems Technologies Pvt Ltd.** Bangalore, Karnataka, India

Membership

- **IEEE** student member
- **Phi Beta Delta** International Honor Society

Publications

- Amar Shrestha, Khadeer Ahmed, Yanzhi Wang, David P. Widemann, Adam T. Moody, Brian C. Van Essen, and Qinru Qiu. "A Spike-Based Long Short-Term Memory on a Neurosynaptic Processor." In progress, *International Conference On Computer Aided Design*. IEEE 2017
- Shrestha, Amar, Khadeer Ahmed, Yanzhi Wang, and Qinru Qiu. "Stable spike-timing dependent plasticity rule for multilayer unsupervised and supervised learning." In *Neural Networks (IJCNN), 2017 International Joint Conference on*, pp. 1999-2006. IEEE, 2017.
- Ahmed, Khadeer, Qinru Qiu, and Mangesh Tamhankar. "Distributed and configurable architecture for neuromorphic applications on heterogeneous cluster." In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pp. 1-7. IEEE, 2016.
- Ahmed, Khadeer, Amar Shrestha, Yanzhi Wang, and Qinru Qiu. "System Design for In-Hardware STDP Learning and Spiking Based Probablistic Inference." In *VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on*, pp. 272-277. IEEE, 2016.
- Ahmed, Khadeer, Amar Shrestha, Qinru Qiu, and Qing Wu. "Probabilistic inference using stochastic spiking neural networks on a neurosynaptic processor." In *Neural Networks (IJCNN), 2016 International Joint Conference on*, pp. 4286-4293. IEEE, 2016.

- Ahmed, Khadeer, Amar Shrestha, and Qinru Qiu. "Simulation of bayesian learning and inference on distributed stochastic spiking neural networks." In *Neural Networks (IJCNN), 2016 International Joint Conference on*, pp. 1044-1051. IEEE, 2016.
- Qiu, Qinru, Zhe Li, Khadeer Ahmed, Wei Liu, Syed Faisal Habib, Hai Helen Li, and Miao Hu. "A neuromorphic architecture for context aware text image recognition." *Journal of Signal Processing Systems* 84, no. 3 (2016): 355-369.
- Ahmed, Khadeer, Qinru Qiu, Parth Malani, and Mangesh Tamhankar. "Accelerating pattern matching in neuromorphic text recognition system using intel xeon phi coprocessor." In *Neural Networks (IJCNN), 2014 International Joint Conference on*, pp. 4272-4279. IEEE, 2014.
- Qiu, Qinru, Zhe Li, Khadeer Ahmed, Hai Helen Li, and Miao Hu. "Neuromorphic acceleration for context aware text image recognition." In *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, pp. 1-6. IEEE, 2014.
- Schlereth, F. H., A. K. Mahabalagiri, A. Khadeer, T. McLoed, J. T. Spencer, and K. S. Sweder. "Frequency measurement for QCM applications." In *Industrial Instrumentation and Control (ICIC), 2015 International Conference on*, pp. 1140-1143. IEEE, 2015.
- Mahabalagiri, Anvith Katte, Khadeer Ahmed, and Fred Schlereth. "A novel approach for simulation, measurement and representation of surface EMG (sEMG) signals." In *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*, pp. 476-480. IEEE, 2011.